

# A component-based approach to distributed system management

## A use case with self-manageable J2EE clusters

Sara Bouchenak, Fabienne Boyer, Emmanuel Cecchet, Sébastien Jean,  
Alan Schmitt and Jean-Bernard Stefani  
*INRIA – Sardes Group*  
*First.Last@inrialpes.fr*

### Abstract

*Clustering has become a de facto standard to scale distributed systems and applications. However, the administration and management of such systems still use ad-hoc techniques that partially fulfill the needs. The expertise needed to configure and tune these systems goes beyond the capacity of a single system administrator or software developer.*

*We present a modular software infrastructure to build command and control loops to manage large scale distributed systems. Our approach uses a reflective component model in a systematic way for building a system model and every single stage in the supervision loop. This approach offers modularity, easy configuration, dynamic reconfiguration, as well as reusability. We illustrate how this architecture can be used to build self-manageable J2EE application server clusters.*

### 1. Introduction

Clusters of commodity hardware are now a common and popular means to provide services with high availability. Indeed, they offer a performance/price ratio that is much better than the one of centralized multiprocessor machines, and they can scale simply by adding new nodes to the system when the need arises.

Such clusters, which may consist of several thousands of nodes – the Google search engine cluster [9] counts more than 20,000 nodes – are indeed large scale distributed systems that are challenging to administer. In fact, the total cost of ownership (TCO) of clusters rises sharply with the cluster size. For instance, our past and ongoing experiences at INRIA with large scale clusters has indicated that a full-time administrator is needed every 200 nodes. At this scale, the mean time between failure (MTBF) is about 1 day: on average, one node crashes (because of hardware or software failure) every day. Hence there is a crucial need for tools that ease the administration of these distributed systems.

Existing tools mainly focus on specific aspects of system management, such as monitoring or deployment. Similarly, complex middleware systems, such as J2EE (Java™ 2 Platform Enterprise Edition) applications, come with proprietary tools for manual configu-

ration based on a static view of the underlying system. The integration of these different tools to manage a given distributed application requires the combined expertise of operating system and database managers, middleware specialists, and end-user application designers. This often results in complex and ad-hoc system management solutions specifically crafted for one particular scenario, with little or no interaction between the different administration tools used. Moreover, such solutions are often brittle: they are difficult to adapt when small changes in the system architecture occur, and it is hard to reuse them when developing other similar applications.

To address these issues, we propose a modular architecture for the management of distributed systems that enables the systematic construction of supervision loops. These loops consist of several stages: *sensors* gather information about the system, this information is carried by *channels* to the decision logic, the *decision logic* chooses what actions to take, these actions are executed by *actuators*, effectively modifying the system state and closing the loop.

In this paper we present a component-based approach to build the different stages of the supervision loop. Such an approach enables modularity, easy configuration, dynamic reconfiguration, and reusability. As most distributed systems require identical sensors, channels, and actuators, we concentrate on the description of these building blocks and how they may be assembled into a supervision loop. Since our approach is not dependent on the specific decision logic used, we leave it unspecified. We also investigate how to leverage this architecture in the building of self-manageable J2EE application server clusters that may be easily configured, deployed, monitored, tuned, and dynamically reconfigured, focusing mainly on efficient resource management.

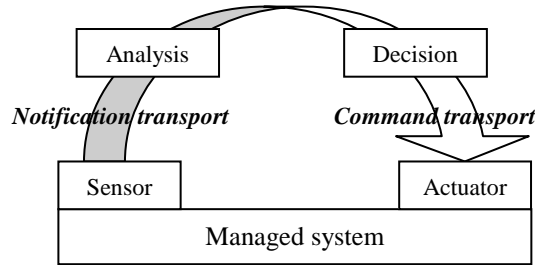
The outline of the rest of this paper is as follows. Section 2 introduces the principles and the overall architecture of our approach. Section 3 details how these concepts can be applied to the administration of J2EE clusters. Finally, section 4 concludes the paper and discusses perspectives.

### 2. Principles and Architecture

Managing a computer system can be understood in terms of the construction of system control loops as

stated in control theory [3]. These loops are responsible for the regulation and optimization of the behavior of the managed system. They are typically closed loops, in that the behavior of the managed system is influenced both by operational inputs (provided by clients of the system), and by control inputs (inputs provided by the management system in reaction to observations of the system behavior).

Figure 1 depicts a general view of control loops that can be divided into multi-tier structures including: *sensors*, *actuators*, *notification transport*, *analysis*, *decision*, and *command transport* subsystems.



**Figure 1. Overview of a supervision loop**

**Sensors** locally observe relevant state changes and event occurrences. These observations are then gathered and transported by *notification transport subsystems* to appropriate observers, i.e., analyzers. The *analysis* assesses and diagnoses the current state of the system. The diagnosis information is then exploited by the *decision* subsystem, and appropriate command plans are built, if necessary, to bring the managed system behavior within the required regime. Finally, *command transport* subsystems orchestrate the execution of commands required by the command plan while *actuators* are the implementation of local commands.

Therefore, building an infrastructure for system management can be understood as providing support for implementing the lowest tiers of a system control loop, namely the sensors/actuators and notification/command transport subsystems. We consider that such an infrastructure should not be sensitive as to how the loop is closed at the top (analysis and decision tiers), be it by a human being or by a machine (in the case of autonomous systems).

In practice, a control loop structure can merge different tiers, or have trivial implementations for some of them (e.g., a reflex arc to respond in a predefined way to the occurrence of an event). Also, in complex distributed systems, multiple control loops are bound to coexist. For instance, we need to consider horizontal coupling whereby different control loops at the same level in a system hierarchy cooperate to achieve correlate regulation and optimization of overall system behavior by controlling separate but interacting subsystems [5]. We also need to consider vertical coupling whereby several loops participate, at different time granularities and system levels, to the control of a system (e.g., multi-level scheduling).

## 2.1. Open challenges

The issues that must be dealt with in the construction of an infrastructure for system management are numerous. We list some of them below.

### 2.1.1. Building a system model

System control and management loops must be anchored at sensors and actuators located at specific coordinates in a system structure. A first issue is thus the provision of an appropriate frame of reference to place sensors and actuators. We call *system model* such a frame of reference. We believe that a system model should be built automatically and consistently (i.e., by maintaining a causal connection between the model and the actual system).

### 2.1.2. Instrumenting the managed system

Implementing sensors and actuators is a second main issue, especially since it seems illusory to foresee all the different forms of observations that may be relevant in the life-time of a complex system. This raises the following issues: (i) dynamic instrumentation, especially in long-lived highly available systems, (ii) instrumentation of legacy software which is bound to be present in complex systems, and (iii) providing instrumentation support to not only capture local events but also causal and temporal relationships which are critical to understand a system behavior (e.g., to counter DDoS attacks, or to diagnose system failures).

### 2.1.3. Implementing transport subsystems

A first issue for these subsystems is scalability with respect to the number of observed events (notification), number of potential targets (commands), and geographical dispersion. Mechanisms for scalability include, for instance, filtering, aggregation, content-based routing, and hierarchical organization of transport channels. A second issue is reliability, i.e., safe notification and command transport to expectant observers and actuators, meeting timeliness requirements, relevance constraints (notifications), and synchronization constraints (commands), even in the presence of failures. A third issue is configurability, to accommodate the diverse settings and environments in which to deploy transport systems, as well as to support the different communication semantics required for notifications and commands.

## 2.2. Component model

To meet these challenges in a principled fashion, we base our architecture and development on a reflective component model.

### 2.2.1. Model characteristics

Before explaining how we use the component model, let us first clarify its main features. The component model we use is the Fractal model [4], which has a number of distinguishing features:

**Explicit binding:** components can interact if they are bound together by means of bindings. A *binding* is a communication path between two or more components. A binding puts in relation component interfaces (i.e., access points to the supporting components).

Interfaces can be either of the server type (to accept incoming method calls) or of the client type (to issue method calls and accept potential returns). A binding can be local or distributed across different machines, and is reified as a component in the model. The Fractal model allows arbitrary forms of bindings.

**Hierarchical components with sharing:** components can be either primitive or composite. Composite components have sub-components. Sub-components can be shared among several different composites.

**Selective reflection:** a component can be endowed, at run-time if necessary, with controllers, that control the execution of the component. A controller can e.g., intercept method calls to a component interface, or export an interface to control some aspects of a component behavior or internal configuration. Controllers are not fixed but the Fractal model identifies several forms of controllers such as content controllers (that allow adding or removing subcomponents in a composite) and life-cycle controllers (that allow stopping or starting the execution of a component).

### 2.2.2. Using components for system management

We use the component model in three main ways. First, the system model is described as a software architecture expressed in terms of the component model, exhibiting bindings between components, containment (e.g., to model subsystems aggregates), and sharing relationships among components (e.g., to model multiplexing of resources among different subsystems). Components are run-time entities in Fractal, so the resulting software architecture mirrors the actual run-time structure. An architecture description language (ADL) can provide an explicit representation of the system model. It should be noted that a system model described as a Fractal component structure exhibits the key property of interaction integrity [12], whereby each component only communicates with other components through its declared client or server interfaces. This is crucial for ensuring isolation and protection properties.

Second, Fractal component controllers provide the basic mechanisms necessary to implement sensors and actuators, at run-time if necessary. Controllers provide essentially the same instrumenting capability as a dynamic aspect weaver such as JAC [6] (e.g., method interception, component state intercession), within a structured system model. In particular, controllers provide us with a way to build behavior monitoring and tracing facilities adapted to each component, to control the lifecycle of components, and to control the way component interfaces are bound and interact. Wrapping legacy software behind a set of component controllers provides us the basic mechanisms for instrumenting it.

Third, the construction of a system with Fractal components provides us with a dynamically reconfigurable structure, which we can exploit to adapt and specialize our infrastructure to various environments, and to provide basic support for adaptations based on system configuration updates (e.g., replacing

faulty components). This is not merely a facility, but a key feature for supporting self-managed systems: as we illustrate below, with the example of automated repair management, a management infrastructure built as an assembly of Fractal components is amenable to the same management policies and operations as the managed system itself.

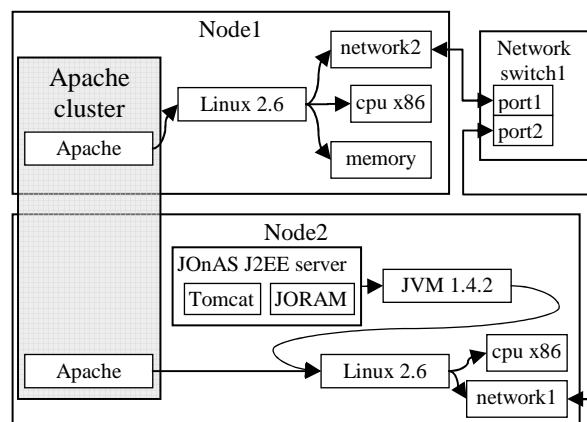
## 3. Autonomous J2EE Clusters

J2EE application servers [1] are usually composed of 4 tiers: a web tier, a presentation tier, a business logic tier, and a database tier. In a J2EE cluster, each tier can be a cluster by itself to provide better fault tolerance and performance scalability. In this section, we explain how the concepts presented in section 2 are used to address the issues that arise in the administration of an application running on a J2EE cluster by analyzing its lifecycle.

### 3.1. Deployment

A J2EE application package contains information about the different components of the application and their configuration. With current J2EE technology, the deployment is described in configuration files that statically map components to resources. Each resource is statically attached to a physical host, thereby tightening the deployment configuration to a specific cluster configuration. Any change in the cluster configuration, due to a node failure for example, will cause the deployment to fail. A manual modification of the configuration file is then needed in order to proceed with a successful deployment.

This process is not acceptable when hosting multiple applications on large clusters that are prone to frequent configuration changes. Hence, there is a need for a cartography service that builds a comprehensive system model, encompassing all hardware and software resources available in the system. The deployment process should dynamically map application components onto available resources by querying the cartography service, that is responsible for maintaining a coherent view of the system state.



**Figure 2. Example of a component representation of a system cartography**

A component-based model is well suited to represent such a system model. Each resource (node, software, ...) can be represented by a component. Composition manifests hierarchical and containment dependencies (e.g., all software components contained in a node fail when the node fails). For example, as depicted in figure 2, a node is a composite component that contains several components, some of which are primitive (CPU, memory, ...) or composite (software containing several pieces of software). Binding between components define interaction relationships. Components can also be shared by different composites to offer different views of the system (an Apache instance running on a node belongs also to the Apache cluster tier).

With such an infrastructure, a deployment description no longer needs to bind static resources but only needs to define the set of required resources. This description might include an exact set of resources or just define a minimal set of constraints to satisfy. The cartography service can then inspect the system representation to find the components that correspond to the resources needed by the application. The deployment process itself consists in inserting the application components into the node component that contains the required resources. Finally, the application components are bound to the resources via bindings to reflect the resource usage in the cartography. The effective deployment of a component is then performed consecutively, as the component model allows some processing to be associated with the insertion or removal of a sub-component.

## 3.2. Monitoring

Once the application has been deployed on the J2EE cluster, we need to know both the system and the application states to be aware of problems that may arise. Most common issues are due either to hardware faults such as a node or network link failure, or inappropriate resource usage when a node or a tier of the application server becomes a bottleneck.

Thus, there is a need for a monitoring service reporting the current cluster state to take the appropriate actions. Such a monitoring infrastructure requires sensors and a notification transport subsystem.

### 3.2.1. Sensors

Sensors can be implemented as components and component controllers that are dynamically deployed to reify the state of a particular resource (hardware or software). Some sensors can be generic to interact with resources through common protocols such as SNMP or JMX/RMI, but other probes are specific to a resource (processor sensor).

Deploying sensors optimally for a given set of observations is an issue. Sensors monitoring physical resources may have to be deployed where the resource is located (e.g., to monitor resource usage) or on remote nodes (e.g., for detecting node failures). Another concern about sensors is their intrusiveness on the

system. For instance, the frequency of probing must not significantly alter the system behavior.

In the case of a J2EE cluster, we have to deal with different legacy software for each tier. Some software, such as web or database servers, do not provide monitoring interfaces, in which case we have to rely on wrapping and indirect observations using operating system or physical resource sensors. However, J2EE containers usually provide JMX interfaces that offer a way to instrument the application server. Additionally, the application programmer can provide user level sensors (e.g., in the form of JMX MBeans).

### 3.2.2. Notification transport

Once the appropriate sensors are deployed, they generate notifications to report the state of the resource they monitor. The notifications must be collected and transported to the observers and analyzers that have expressed interest in them. An observer can for instance be a monitoring console that will display the state of the system in a human readable form. Different observers and analyzers may require different properties from the channel used to transport the notifications. An observer in charge of detecting a node failure may require a reliable channel providing a given QoS, while these properties are not required by a simple observer of the CPU load of a node. Therefore the channels used to transport the notifications should be configured according to the requirements of the concerned observers and analyzers. Typically, it should be possible to dynamically add, remove, or configure channels.

To this effect, we have implemented DREAM (Dynamic Reflective Asynchronous Middleware) [2], a Fractal-based framework to build configurable and adaptable communication subsystems, and in particular asynchronous ones. DREAM components can be changed at runtime to accommodate new needs such as reconfiguring communication paths, adding reliability or ordering, inserting new filters, and so on. We are currently integrating various mechanisms and protocols in the DREAM framework to implement scalable and adaptable notification channels, drawing from recent results on publish-subscribe routing, epidemic protocols, and group communication.

## 3.3. Reconfiguration

Once a decision has been made (e.g., extension of a J2EE tier on new nodes to handle increased load), the decision tier is responsible for issuing a command workflow for implementing the decision. In general, the workflow logic corresponds to some system reconfiguration, that is orchestrated by the command transport subsystem and is performed by actuator components. Command channels that transport commands can be built using the same technology as the one used for notification channels but require additional properties such as causal ordering, atomicity, or transactional behavior.

We are currently extending the DREAM framework to add these non-functional properties to build com-

mand channels with DREAM components. Executing the workflow logic can rely on transactional workflow technology [10].

### 3.4. Supervision loops use cases

This section presents how the infrastructure we propose can be used in two J2EE clusters use cases: admission control and automated repair management.

#### 3.4.1. Admission control

If a request flow is higher than the capacity of a server in charge of its processing, the server will be overloaded and may crash. Each server instance must have its own supervision loop to regulate its load by tuning the admission control level. Examples of admission control policies can be setting the maximum number of client connections for Web or database servers, or tuning pool sizes in an EJB container.

In this case, a *local* control loop is associated with each server instance and acts as follow:

- *sensor*: reports local load that can be an aggregation of several sensor components like CPU, memory, network and disk IO usage. In this case the sensor component can be a composite made of the specific sub-sensors.
- *notification channel*: alarms are triggered by the sensor component when a lower/upper limit is reached. These events are forwarded to the analysis component through the notification channel component, seen as a connector.
- *analysis/decision component*: processes the incoming events and updates admission control settings through a command channel component, seen as a connector.
- *actuator*: updates the server configuration and activates it.

#### 3.4.2. Repair management

Machine and software failures are bound to occur often in a large J2EE cluster. While several tiers in current J2EE technology have built-in failover features, the repair and recovery of failed components remains a largely manual operation. To automate repair management, we can build a *global* control loop that supports a repair management policy, triggered upon the occurrence of a failure. We illustrate this with a simple policy and its implementation using our architecture.

*System model.* The system model for repair management corresponds to a component-based description of the managed system, where failure dependencies are modeled as standard component containment relationships. In particular, we just consider node and J2EE components. A node component is an abstraction of the behavior of the operating system and its supporting hardware. Failure of a node component (stopping failure) implies the correlated failure of all supported J2EE components. J2EE components represent abstractions of the different elements in a J2EE tier, together with their supporting Java virtual machine (if any) and operating system process. We consider that the failure mode of J2EE component is fail-

stop (this would in theory require appropriate instrumentation in wrappers). As a simplifying assumption, we consider that all nodes are fully interconnected by a non-faulty network. The resulting system model is thus composed of :

- a set of nodes, that correspond to the scope of the repair management function. All nodes within the scope of the function are subject to the same management policy, i.e. they form part of a repair management *domain*.
- within each node, a set of J2EE components, that are also part of the repair management domain.
- between J2EE components, a set of component bindings, corresponding to the nominal connections between tiers. As a simplifying assumption, we assume that the failure of a J2EE component, or of its supporting node, leads to the failure of all the bindings in which it participates (stopping failure).

In addition to above assumptions on failures, we consider the following behavior:

- bootstrapping a machine creates a node component. Each node component is equipped with a binding factory component enabling the creation of bindings between different nodes. Once a node is running, it is possible to bind this node to any other node within the repair management domain (this implies, as per the domain abstraction, that the repair management domain is a naming context and communication domain).
- each node component provides the ability to create and delete J2EE components. Each J2EE component provides for the ability to create and delete bindings between J2EE components, possibly residing in different nodes.

*Sensors.* Observations required for repair management include: the occurrence of a node failure and the occurrence of a J2EE component failure. We consider that the failure of a J2EE component can be observed locally, i.e. that each node component automatically detects the failure of any J2EE component it contains. Sensors for detecting node failures must involve some form of distributed failure detection protocol.

*Actuators.* Actions required in response to a failure vary according to the failed component. For J2EE components, actions involve removing dead bindings, starting new J2EE components, and binding them to other running J2EE components. For node components, actions involve removing dead bindings, node bootstrapping, installing J2EE components, establishing bindings to other J2EE components. All the actions above are actions pertaining to the control of components (esp. content and life-cycle management in Fractal terms).

*Transport.* Failures are notified to the analysis stage on occurrence. Notification of failures follow an asynchronous operation semantics, but are subject to the timing and safety guarantees (e.g. notification should take place within a specified delay bound; notification

loss should be below a given threshold). Commands requiring the execution of actions obey a synchronous operation at most once semantics, under timing constraints.

*Policy (analysis and decision).* The repair management policy adopted in this paper (for illustration purposes) is extremely simple:

- on receipt of a J2EE component failure, execute the following commands, in sequence: terminate bindings to the failed component; start a new J2EE component replacing the previous one in the same node; re-establish failed bindings.
- on receipt of a node failure, execute the following commands in sequence: terminate bindings to all the failed J2EE components which were supported by the node; elect a new node; install an isomorphic configuration to the failed one on the selected node; re-establish all failed bindings to the new configuration.
- for node election, one can apply for instance the following failover scheme: allocate to each tier a given subset of the total set of nodes; partition each subset in two sets: active nodes in the tier and failover nodes in the tier. When an active node fails in a tier, choose an arbitrary failover node in the same tier as the elected node.

The above policy implicitly considers stateless configurations, i.e. it assumes that there is no shared state information which is maintained across the different tiers and their associated components. This may not be true in practice, e.g. with stateful session beans or entity beans. Also, the policy above is minimal in the sense that it considers all external requests and their associated sessions are equivalent. This also may not be true in practice, e.g. because of differentiated services granted to sessions according to age. The repair management policy can be implemented simply on a single node. However, the possible failure of the node running the policy stage should be dealt with. As a simple solution, one can consider a self-managed loop, whereas the policy stage is implemented as a replicated service on distinct nodes, e.g. using an active replication scheme. Failure and subsequent repair of a node running the policy stage can thus be handled automatically using, *mutatis mutandis* (with J2EE components replaced by policy components), the same repair management policy than described above.

## 4. Conclusion

We have presented a modular architecture for the principled and component-based construction of system management functions. We have illustrated the use of the architecture in an important practical use case, the management of J2EE application clusters. Compared to the industrial state of the art, which relies on simple manager/agent monitoring loops based on JMX and SNMP, our architecture allows the construction of system-wide, multi-level observations and reconfiguration actions. Furthermore, through a com-

bination of loop composition and scalable transport technology, we expect our architecture to scale to different system sizes and organizations.

Managing computational resources in computer clusters or grids is the subject of intense ongoing research activity [7, 8]. These works tend to concentrate on the construction of various schemes for automatic resource management in networked sets of computational resources. In contrast, we describe an architecture that can be used to implement all the classical system management areas (resources, failures, configurations, security, costs). Such resource management schemes can be properly folded in our architecture (essentially, they can be seen as providing different decision tiers in our framework, and relying on specific system models). A work which is close to ours in principle is [11], which combines monitoring and workflow facilities to manage the execution of applications in a PC cluster. Compared to this work, we define a scalable architecture for implementing complete control loops and relying on a component model for a systematic approach to dynamic reconfiguration.

We are currently working on several open issues for the implementation of our architecture: system model and instrumentation for resource accounting and allocation, scalability issues in the notification transport subsystem, coordination in presence of failures in the command transport subsystem, and automating the analysis and decision tiers for our J2EE use cases.

## 5. References

- [1] S. Allamaraju et al. – Professional Java Server Programming J2EE Edition - Wrox Press, 2000.
- [2] V. Quéma et al. – DREAM: a Software Framework for the Construction of Asynchronous Middleware – *INRIA research report*, 2004.
- [3] K. Ogata – Modern Control Engineering, 3<sup>rd</sup> ed. – Prentice-Hall, 1997.
- [4] E. Bruneton et al. – An Open Component Model and its Support in Java – *Proceedings of ISCSE*, 2004, to appear.
- [5] G. Goldszmidt, Y. Yemini. – Distributed Management by Delegation - *Proceedings of ICDCS-15*, 1995.
- [6] R. Pawlak et al. – JAC: A Flexible Solution for Aspect-Oriented Programming in Java – *Proceedings of Reflection* 2000.
- [7] Y. Fu et al. – SHARP: An architecture for secure resource peering – *Proceedings of SOSP'03*, 2003.
- [8] M. Aron et al. – Cluster reserves: a mechanism for resource management in cluster-based network servers – *Proceedings of SIGMETRICS'00*, 2000.
- [9] L. Barroso et al. – Web Search For A Planet: The Google Cluster Architecture – *IEEE Micro*, vol. 23, 2003.
- [10] I. Houston et al. – The CORBA Activity Service Framework for Supporting Extended Transactions - *Software: Practice and Experience*, Volume 33, Issue 4, 2003.
- [11] W. Bausch et al. – BioOpera: Cluster-aware Computing – *Proceedings of 4<sup>th</sup> IEEE Cluster*, 2002.
- [12] S. White, J. Hanson, et al. – An Architectural Approach to Autonomic Computing – *Proceedings of IEEE Int. Conf. On Autonomic Computing (ICAC 04)*, 2004.