Autonomic Management for Grid Applications

Mohammed Toure [1], Girma Berhe [1], Patricia Stolf [2], Laurent Broto [3]
Noel Depalma [4], Daniel Hagimont [1]

[1]*IRIT, Institut National Polytechnique de Toulouse, CNRS ; 118 Route de Narbonne, 31062 Toulouse France*
[2]*IRIT, Institut Universitaire de Formation des Maitres de Midi-Pyrenees, CNRS*
118 Route de Narbonne, 31062 Toulouse France
[3]*IRIT, CNRS ; 118 Route de Narbonne, 31062 Toulouse France*
[4]*INRIA, Institut National Polytechnique de Grenoble*
655 avenue de l'Europe 38330 Montbonnot - St Martin France
[1], [2], [3]`First.Last@irit.fr` [4]`First.Last@inrialpes.fr`

Abstract

Distributed software environments are increasingly complex and difficult to manage, as they integrate various legacy software with specific management interfaces. Moreover, the fact that management tasks are performed by humans leads to many configuration errors and low reactivity. This is particularly true in medium or large-scale grid infrastructures. To address this issue, we developed Jade, a middleware for self-management of distributed software environments. In this paper, we report on our experiments in using Jade for the management of grid applications.

1 Introduction

Today's computing environments are becoming increasingly sophisticated. They involve numerous complex software that cooperate in potentially large-scale distributed environments. These software are developed with very heterogeneous programming models and their configuration interfaces are generally proprietary. The management of these software is a very complex task achieved by human administrators.

A very promising approach to the above issue is to implement administration as an autonomic software. Such a software can be used to deploy and configure applications in a distributed environment. It can also monitor the environment and react to events such as failures or overloads and reconfigure applications accordingly and autonomously. There are different advantages to this approach. First, it provides a high-level support for deploying and configuring applications reduces errors and administrator's efforts. Secund, autonomic management allows the required reconfigurations to be performed without human intervention, thus saving administrators' time. And then, autonomic management is a means to save hardware resources as resources can be allocated only when required (dynamically upon failure or load peak) instead of pre-allocated. In this vein, we propose Jade, an environment for developing autonomic management software. Jade mainly relies on the following features:*A component model*: Jade models the managed environment as a component-based software architecture which provides means to deploy, configure and reconfigure the software environment. *Control loops* which link probes to reconfiguration services and implement autonomic behaviors. We used Jade for deployment and autonomic management of grid applications. Specifically, we implemented self-optimization and self-repair for different grid applications. The rest of the paper is organized as follows. Section 2 presents the two case studies that we consider in this paper : a clustered enterprise application and a grid computing application. Section 3 presents the design principles of Jade. Autonomic management policies for the case studies are presented in Section 4. Results of our experimental evaluation are presented in Section 5, then related work is discussed in Section 6. Finally, Section 7 draws our conclusions.

2 Case Studies

2.1 Clustered J2EE applications

The Java 2 Platform, Enterprise Edition (J2EE) defines a model for developing web applications [24] in a multi-tiered architecture. Such applications are typically composed of a web server, an application server and a database server.

Upon an HTTP client request, either the request targets a static web document, in which case the web server directly returns that document to the client; or the request refers to a dynamic document, in which case the web server forwards that request to the application server. When the application server receives a request, it runs one or more software components (e.g. Servlets, EJBs) that query a database through a JDBC driver (Java DataBase Connection driver) [25]. Finally, the resulting information is used to generate a web document on-the-fly that is returned to the web client.

In this context, the increasing number of Internet users has led to the need of highly scalable and available services. To face high loads and ensure availability of Internet services, a commonly used approach is the replication of servers in clusters. Such an approach usually defines a particular software component in front of each set of replicated servers, which dynamically balances the load among the available replicas. An example of clustered J2EE architecture is shown in Fig. 1 where Apache web servers are connected to Tomcat servlet servers (through mod_jk connectors), themselves connected to Jonas EJB servers (through CMI connectors), themselves connected to MySQL database servers (through JDBC connectors).

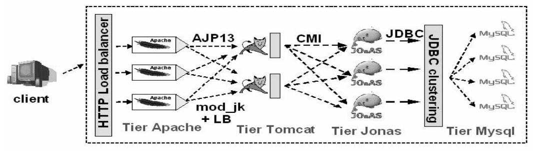

Figure 1. A clustered J2EE architecture

This represents an interesting experimental environment since it brings together most of the challenges addressed by Jade: *the management of a variety of legacy systems*,since each tier in the J2EE architecture embeds a different piece of software (e.g.a web server, an application server, or a database server), *very complex administration interfaces and procedures* associated with very heterogeneous software (each server is parameterized through several configuration files whose format is proprietary), *the requirement for high reactivity* in taking into account events which may compromise the normal behavior of the managed system, e.g. load peaks or failures.

2.2 Grid Computing applications

Grid computing is a type of parallel and distributed computing that enables the sharing, selection, and aggregation of geographically distributed resources, dynamically at runtime depending on their availability, capability, cost and user's quality of service requirements [7] .

DIET [11] is an example of middleware environment which aims at balancing computation load in such a grid.The aim of DIET (Fig. 2) is to provide transparent access to a pool of computational servers at a very large scale.

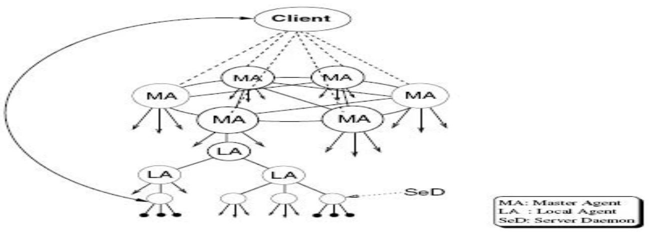

Figure 2. DIET Architecture

DIET mainly has the following components: client, Master Agents (MA), Local Agents (LA) and Server Daemons (SeD). A client is an application which uses DIET to solve problems. Master Agents (MA) receive computation requests from clients. Then a MA chooses the best server and returns its reference to the client. The client then sends the computation request to that server. Local Agents (LA) aim at transmitting monitoring information between servers and MAs. Server Daemons (SeD) encapsulate computational servers (processors or clusters). A SeD declares the problems it can solve to its parent LA and provides an interface to clients for submitting their requests.

The DIET middleware also represents an interesting example for our autonomic management environment, since it brings together many of the challenges addressed by Jade.
The management of a distributed organization of legacy software (MA, LA and SeD).
Distributed configuration, since DIET requires the configuration of these software (through a set of configuration files) to implement a consistent hierarchical structure.
Self-repair : due to several reasons (hardware or software), a server may stop functioning. The goal is to detect the failure and repair it automatically.

3 Jade Design Principles

Since Jade addresses the management of very hererogeneous software, our key design choice was to rely on a component model to provide a uniform management interface for any managed resource. Therefore, each legacy software is wrapped in a component which interfaces its administration procedures.

We first present in Section 3.1 the component model (Fractal [6]) used in our system. We then describe in Section 3.2 the wrapping of software in Fractal components. Section 3.3 presents how applications can be deployed, thanks to this component-based approach.

3.1 The Fractal Component Model

The Fractal component model intends to implement, deploy, monitor, and dynamically configure complex software systems, including operating systems and middleware. This motivates the main features of the model: composite components (to have a uniform view of a software environment at various levels of abstraction), introspection capabilities (to observe the architecture of a deployed system), and reconfiguration capabilities (to dynamically configure a system).

A Fractal component is a run-time entity that is encapsulated with one or more interfaces. Interfaces are access point to a component, they can be of two kinds: server interfaces, which correspond to access points accepting incoming method calls, and client interfaces, which correspond to access points supporting outgoing method calls. The signatures of both kinds of interface can be described by a standard Java interface declaration, with an additional role indication (server or client). A Fractal component can be composite, e.g. defined as an assembly of several subcomponents, or primitive, e.g. encapsulating an executable program.

Communication between Fractal components is only possible if their interfaces are bound.

The originality of the Fractal model lies in its open reflective features. In order to allow for well scoped dynamic reconfiguration, components in Fractal can be endowed with controllers, which provide access to a component internals, allowing for component introspection and the control of component behaviour.

A controller provides a control interface and implements a control behavior for the component, such as controlling the activities in the components (suspend, resume) or modifying some of its attributes. Useful forms of controllers are specified by the model. They can be combined and extended to yield components with different control features, including the following: *Attribute controller* : an attribute is a configurable property of a component. This controller supports an interface to expose getter and setter methods for its attributes.
Binding controller :supports an interface to allow binding and unbinding its client interfaces to server interfaces by means of primitive bindings.
Content controller: for composite components, supports an interface to list, add and remove subcomponents in its contents.
Life-cycle controller: this controller allows an explicit control over a component execution. Its associated interface includes methods to start and stop the execution of the component.

Several implementations of the Fractal model have been issued in different contexts. The work reported in this paper relies on an implementation on top of the Java virtual machine: Julia.

3.2 Component-based management

The main idea of Jade is to wrap each legacy software in a Fractal component so that the overall software environment can be managed by programs (instead of humans). These programs can rely on the controllers provided by the fractal components to observe or modify the administrated software environment. The benefits from reifying legacy software in components are:*Managing legacy entities using a uniform model* (the Fractal control interface), instead of relying on software-specific, hand-managed, configuration files, *Managing complex environments with different points of view* : for instance, using appropriate composite components, it is possible to represent the network topology, the configuration of a J2EE middleware, or the configuration of a DIET architecture, *Adding a control behavior to the encapsulated legacy entities*(e.g. monitoring, interception and reconfiguration).

Therefore, the Fractal component model is used to implement a management layer on top of the legacy layer (composed of the actual managed software). Fig. 3 represents a generic architecture for clustered applications). Each legacy entity (LE) is wrapped by a Fractal component (FC). The vertical dashed arrows (between the management and legacy layers) represent management relationships between fractal components (FC) and the wrapped software entities/legacy entities(LE). In the legacy layer, the dashed lines represent relationships (or bindings) between legacy entities, whose implementations are proprietary. These bindings are represented in the management layer by (Fractal) component bindings (full lines in the figure). The number of bindings that an entity can have depends on the applications configuration and particularity.

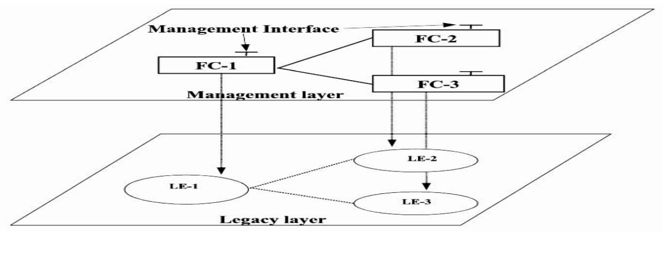

Figure 3. Management layer for clustered applications

Each legacy entity (server) of the application is wrapped in a Fractal component. For example in DIET, LE-1 is a MA, LE-2 and LE-3 are LAs. In J2EE, LE-1 is an apache web server, LE-2 and LE-3 are Tomcat servlet servers. Each

FC provides an attribute controller, a binding controller and a lifecycle controller with the following functionalities:

The attribute controller interface is used to set/get attributes of the FC. A modification of the attribute of a FC is reflected to the equivalent attribute in the associated LE. For instance, if LE-1 is an Apache server, a modification of the port attribute is reflected to the httpd.conf file (which includes a *port* variable).

The binding controller interface is used to set up a binding between two FCs. This binding is reflected at the legacy layer on the associated legacy entities. For example, for J2EE applications, the binding between Apache and Tomcat servers is reflected in the worker.properties file.

The life cycle controller interface is used to start or stop a FC. Its implementation depends on the application. For example in DIET, starting a FC implies starting a server daemon (an executable file) in the legacy layer.

Once wrappers have been implemented for a given environment, the management layer allows the implementation of sophisticated administration programs which can: introspect (observe managed components) and reconfigure (change component attributes or bindings between components) component architectures. An administration program can add or remove a server replica in the application infrastructure to adapt to workload variations (self-optimization policy) or to adapt to server failure (self-repair policy).

3.3 Deployment

The architecture of an application is described using an Architecture Description Language (ADL), a key feature of the Fractal component model. This description is an XML document which details the architectural structure of the application to deploy. Since the legacy layer is reified in a component based management layer, Fractal ADL can be used to deploy the software infrastructure in a grid, i.e. to describe which software resources compose the grid application, how many servers and/or replicas are created, how are the servers bound together, etc ...

A Software Installation Service component (a component of Jade) allows retrieving the encapsulated software resources involved in the grid application (e.g. an Apache Web server software, MySQL database server software, MA, SeD, etc.) and installing them on the target machines. A Machine Allocation Service component is responsible for the allocation of nodes (from a pool of available nodes) which will host the servers.

The deployment of an application is the interpretation of an ADL description, using the Software Installation Service and the Machine Allocation Service to deploy application's components on nodes. The autonomic administration software (Jade) is also described using this ADL and deployed in the same way.

4 Autonomic Management Policies

Many different autonomic management policies may be implemented, among others: self-configuration, self-optimization, self-healing and self-protection. The management polices considered in this paper are self-optimization and self-repair. The self-repair policy presented here is generic and can be used for the two considered applications (J2EE and Diet) with little or no modification. Unlike self-repair, self-optimization is usually affected by the application's behavior and the optimization target (performance, resource usage, energy, etc.). The self-optimization policy that we describe in this paper aims at providing scalability of a J2EE clustered application. A different and specific algorithm should be proposed to deal with performance issues in a Diet infrastructure.

Autonomic management is achieved through autonomic managers which implement feedback control loops (Fig. 4). These loops regulate, optimize and repair the behavior of the managed system. Each autonomic manager in Jade is based on a control loop that includes sensors (for monitoring the managed system), actuators (which implement reconfiguration procedures) and analysis/decision components (defining a policy).

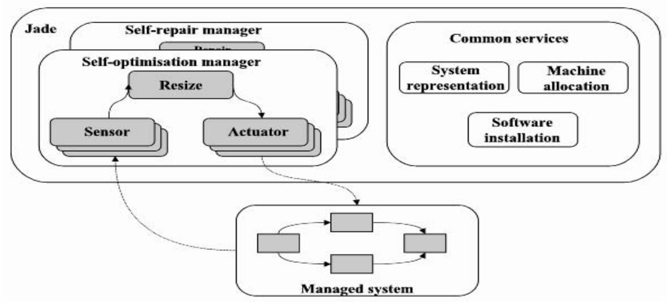

Figure 4. Control loops in Jade

4.1 Implementation of Self-Optimization for the J2EE application

Here, we consider implementing self-optimization using resizing techniques, i.e. dynamically increasing or decreasing the number of nodes allocated to the application.

A standard pattern for implementing scalable clustered servers is the load balancer. In this pattern, a given application server is statically replicated at deployment time and a front-end proxy (the load balancer) distributes incoming requests among the replicated servers.

Jade aims at autonomously adjusting the number of replicas used by the application when the load varies. This is implemented by an autonomic manager which implements the required control loops. We implemented two control loops, one devoted to the management of the replicated web container (Tomcat) and the other devoted to the management of the replicated database (MySQL). In both cases the control loop has the following sensors, actuators and reactors.

Sensors: sensors periodically measure the chosen performance (or QoS) criteria, i.e. a combination of CPU usage and user-perceived response time. In our expriments with Tomcat and MySQL servers, the used sensor is a probe that collects CPU usage information on all the nodes where such a server is deployed. This probe computes a moving average of the collected data in order to remove artifacts characterizing the CPU consumption. It finally computes an average CPU load across all nodes, so as to observe a general load indication of the whole replicated server.

Actuators: the actuators are used to reconfigure the system. Thanks to the uniform management interface provided by Jade, most of the code of the actuators is generic, since increasing or decreasing the number of replicas of an application is implemented as adding or removing components in the application structure.

Reactors: They implement an analysis/decision algorithm. They receive notifications from sensors, and react, if needed, by increasing or decreasing the number of replicas allocated to the controlled tier. In our experiments, the decision logic implemented to trigger such a reconfiguration is based on thresholds on CPU loads provided by sensors.

The main operations performed when more replicas are required are the following: allocate free nodes for the application, deploy the required software on the new nodes if necessary, perform state reconciliation with other replicas in case of servers with dynamic data (see details below), and integrate the new replicas with the load balancer. Similarly, if the clustered server is under-utilized, the main operations performed are the following: select and stop some replicas, unbind them from the load balancer, and release the hosting nodes. To create an additional replica (i.e. node + software server), Jade provides common services which can be invoked from actuators for node allocation and software installation.

One important issue to address when managing replicated servers with dynamic data (modifiable state) is data consistency. This is not a problem in the case of the web container (Tomcat) as our evaluation application is composed of servlets with no dynamically changing information (servlet session state). In the case of database servers, the load balancer that we used is c-jdbc; c-jdbc plays the role of load balancer and replication consistency manager [10], each server containing a full copy of the whole database (full mirroring).

To manage a dynamic set of database servers, a newly allocated server must synchronize its state with respect to the whole clustered database before it is activated. To do so, a ”recovery log” has been added to the c-jdbc load-balancer. It keeps trace of all the requests that affect the state of the database. When a new server is inserted in the clustered database, the recovery log is used to replay modifications (since to previous exit from the cluster) to resynchronize the server before insertion.

4.2 Implementation of Self-repair: Generic

We consider fail-stop machine crashes. Repair management aims at restoring the managed system to a state equivalent to the one that was prior to failure. Whenever a node crash is detected, the autonomic manager must allocate a replacement machine, create on this machine the same components that were located on the faulting node, consistently rebind the architecture to take into account the replacement components, and finally restart some of the components to take into account the modifications on the application’s architecture.

The control loop of this autonomic manager has the following sensors, actuators and reactors.

Sensors: a set of sensors are deployed in order to detect machine failures. These sensors rely on periodic probing requests (heartbeat). Whenever a failure is detected by a sensor, an event is delivered to a reactor.

Reactors: in our experiment, we implemented a single reactor which simply launches the execution of an actuator for reparing the application architecture. Different reactors could be implemented to deal with different types of failures, therefore invoking different sequences of actuators.

Actuators: as for the self-optimization autonomic manager, most of the code of the actuators is generic. We implemented an actuator which allocate a new node and recreate components equivalent to the components that were located on the faulting node, and rebind these components to obtain an equivalent overall architecture.

To repair the software architecture, the actuator has to inspect the components of the architecture (types, attributes, bindings) in order to reproduce a setting equivalent to the one preceeding the failure. However, if a node hosting a server has crashed, the Fractal component encapsulating that server is lost. For this reason, a System Representation service is necessary. The System Representation service maintains a representation of the current architectural structure of the managed system, and is used for failure recovery. This representation reflects the current architectural structure of the system (which may evolve) and it is reliable in the sense that it is itself replicated to tolerate faults. The System Representation is implemented as a meta-level Fractal architecture.

5 Evaluation

The application cases described in section 2 have been implemented and Jade was used to deploy and administrate them. The evaluations reported in this paper focus on self-optimization for the database tier of a J2EE application and self-repair for DIET. A more detailled description of self-optimization and self-repair implementation for J2EE applications can be found in [16] and [5] respectively.

5.1 J2EE: Self-optimization (at database tier)

The experiment was conducted on a cluster X86-compatible machines connected through a 100Mbps Ethernet LAN running various versions of the Linux Kernel. The RUBiS application benchmark [2] is used as J2EE application. RUBiS implements an eBay-like auction system and includes a workload generator. Our J2EE software environment relies on Jakarta Tomcat 3.3.2 (Web and servlet servers) [26], MySQL 4.0.17 (database server) [20], C-JDBC 2.0.2 (database load-balancer) [10], PLB 0.3 (application server load-balancer) [21], Sun's JVM JDK 1.5.0.04 and MySQL Connector/J 3.1.10 JDBC driver (to connect the database load-balancer to the database servers). The machines are distributed as follows: one node for the Jade management platform, one node for the PLB load-balancer, up to two nodes for replicated Tomcat servers, one node for the C-JDBC load-balancer, up to three nodes for the replicated MySQL servers, one node for the RUBiS workload generator. During execution the number of involved machines varies according to the workload.

To evaluate the effectiveness of Jade management platform, we designed a scenario which illustrates the dynamic allocation and deallocation of nodes to tackle performance issues related to a changing workload: (a) at the beginning of the experiment, the managed system is submitted to a medium workload (80 clients) then (b) the load increases progressively up to 500 clients (21 new emulated clients every minute), finally (c) the load decreases symmetrically down to the initial load (80 clients).

To quantify the effect of the reconfiguration, the scenario has been experimented without Jade, that is without any reconfiguration, so that the managed system is not resized. Fig. 5 presents the result of this experiment on the database tier.

The two horizontal lines represent the thresholds used to trigger dynamic reconfiguration (insertion or removal of a database replica). The stair-like curve indicates the number of database replicas.

Without Jade, the CPU usage rapidly saturates, which results in a trashing of the database. With Jade, when the average CPU usage reaches the maximum threshold, the man-

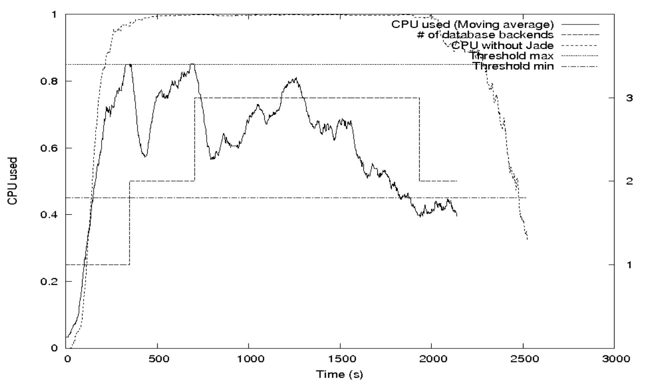

Figure 5. Behavior of the database tier

ager triggers the deployment of a new database server which implies a decrease of the average CPU usage. Symmetrically when the average CPU usage gets under the minimum threshold, the manager triggers the removal of one server.

5.2 Diet: Self-repair

The experiment was conducted in a cluster environment similar to that of Section 5.1. In this experiment we used Jade to deploy a DIET architecture composed of one MA, one LA and two SeDs. The main objective of this experiment is to demonstrate the effectiveness of automatic repair in the case of server failure. Consequently, we artificially induced the crash of a server in the managed system and we observed the load distribution (the CPU usage) on the different servers.

The different machines were distributed as follows: 1 machine for the MA, 1 machine for the LA, 2 machines for the SeDs (computing servers), 1 machine for the Jade management platform, 3 machines for submitting client requests (each machine sends 5000 requests). Our experiment uses a DGEMM [1] computation of 100*100 matrix.

Fig. 6 shows the observed behaviour without using Jade. Both servers (SeD1 and SeD2) have a CPU usage [2] which stabilizes at approximatively 50%, until the crash of SeD2. After the crash, the workload is totally sent to the single remaining server(SeD1), which CPU usage increases rapidly up to 80%.

Fig. 7 shows the observed behaviour with Jade's repair management. During the interval between the crash and the repair, the CPU usage of server SeD1 increases rapidly since the workload is sent to the single remaining server (SeD1), but only for a short time interval (about 10 seconds), as Jade detects the failure and replaces the failed server by a new server (SeD3). Rapidly, the two servers

[1]DGEMM: a matrix computation which is part of linear algebra problems. It is often used as a point of reference for performance evaluation.

[2]the reported CPU usage is an average over 5 seconds.

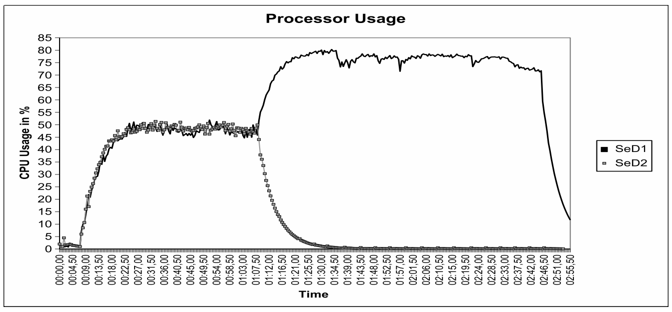

Figure 6. Failure scenario without Jade

(SeD1 and SeD3) stabilize at the same CPU usage level as before the crash.

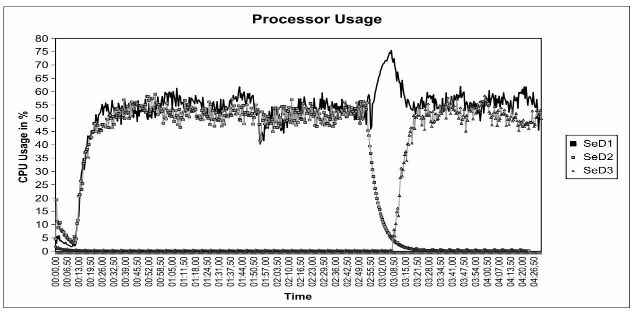

Figure 7. Self-repair: Self-repair with Jade

6 Related work

6.1 Autonomic computing

Autonomic computing is an emerging technology which aims at providing systems with the ability of managing themselves (self-management) with minimum or no human intervention. Several research works have been conducted in addressing the challenges of autonomic computing. As a result, different autonomic systems have been developed. These systems can be broadly classified as (1) systems that incorporate autonomic mechanisms for problem determination, monitoring, analysis, management, etc.(e.g. OceanStore [19], Oceano [3], AutoAdmin [4], Q-Fabric [8]), (2) systems that investigate models, programming paradigms and development environments to support the development of autonomic systems and applications (e.g. Kx(Kinesthetics eXtrem) [18], Astrolable [23], Autonomia [17], AutoMate [1]).

The Jade system falls in the second category and Kx is closely related to it. Both Jade and Kx provide autonomic capabilities onto legacy systems, by relying on specific Actuators (Effectors in Kx) to adapt a legacy software. In the Jade framework, a legacy system is wrapped into a Fractal component. The Fractal model provides uniform management interfaces that enable the implementation of complex administration tasks. This approach makes Jade generic, since once legacy software have been wrapped, their administration can be based on the uniform component model.

6.2 Grid deployment and management

Other work has been done in the Grid world for general application deployment and management. IBM's OptimalGrid [12] supports instrumentation and self-optimization in a Grid computing environment, but requires a customized implementation which it can then assemble and coordinate as opposed to supporting legacy software. Similarly, the AutoMate project [1] defines autonomic system components as being built on a particular framework known as Distributed Interactive Object Substrate (DIOS), and tightly controls execution of these components. In contrast, Jade does not rely on a particular application programming model and it responds to the OGSA's administrative cost reduction requirement [14] . GoDIET [9] is a tool usable to deploy DIET system but does not provide autonomic management such as Jade. The Globus toolkit [13] is a grid middleware providing remote process management, resources discovery and many others services. Jade is not a grid middleware, it aims at providing support for autonomic administration of legacy software environments, as was demonstrated with DIET and J2EE. Research results have been produced to offer autonomic computing to web services. [22] proposes an autonomic Web Services Resource Framework (WSRF). [15] presents a self-healing autonomic attribute implementation using Web services to overcome the problem of heterogeneous computing systems. Our work does not only focus on Web services, it offers autonomic management policies to different legacy software with an encapsulation into Fractal components.

7 Conclusion

In this paper we have presented the design of a framework for the development of autonomic administration services. This framework called Jade relies on the encapsulation of legacy software pieces into Fractal components, thus providing adequate support for programming autonomic services. We presented two case studies (J2EE and DIET) where Jade can be advantageously applied. By wrapping legacy software into Fractal components, autonomic management policies such as self-optimization or self-repair can be implemented using control interfaces (attribute, binding, life-cycle) provided by the Fractal model.

Our future work will focus on deployment and autonomic administration of DIET in large-scale distributed environments (Grid) involving thousands of machines, as in the french Grid'5000 initiative. This will require the adaptation of Jade in large scale environment. This will involve the management of many deployment and administration points, which should lead to a distributed hierarchical Jade system. The work reported in this paper benefited from the support of the French National Research Agency through projects Selfware (ANR-05-RNTL-01803) and Lego (ANR-CICG05-11).

References

[1] M. Agarwal, V. Bhat, Z. Li, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, M. Parashar, B. Khargharia, and S. Hariri. AutoMate: Enabling Autonomic Applications on the Grid. In *Autonomic Computing Workshop - 5th Annual International Workshop on Active Middleware Services*, Seattle, WA, June 2003.

[2] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and Implementation of Dynamic Web Site Benchmarks. In *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, Nov. 2002.

[3] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, and M. Kalantar. Oceano - SLA based management of a computing utility. In *7th IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001.

[4] AutoAdmin. AutoAdmin: Self-Tuning and Self-Administering Databases. http://research.microsoft.com/dmx/autoadmin/.

[5] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. Depalma, V. Quema, and J.-B. Stefani. Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, Orlando, Florida, Oct. 2005.

[6] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *International Workshop on Component-Oriented Programming (WCOP-02)*, Malaga, Spain, June 2002. http://fractal.objectweb.org.

[7] R. Buyya and S. Venugopal. A Gentle Introduction to Grid Computing and Technologies. In *CSI Communications, pages 9-19, Vol. 29*, Mumbai, India, July 2005.

[8] C. Poellabauer. Q-Fabric - System Support for Continuous Online Quality Management. http://www.cc.gatech.edu/systems/projects/ELinux/qfabric.html.

[9] E. Caron, P. K. Chouhan, and H. Dail. GoDIET: A Deployment Tool for Distributed Middleware on Grid'5000. In *EXPGRID workshop. Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Apllications and Tools. In conjunction with HPDC-15*, Paris, France, June 2006.

[10] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference, Freenix track*, Boston, MA, June 2004. http://c-jdbc.objectweb.org/.

[11] P. Combes, F. Lombard, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In *7th Asian Computing Science Conference*, Jan. 2002.

[12] G. Deen, T. Lehman, and J. Kaufman. The Almaden OptimalGrid Project. In *Autonomic Computing Workshop - 5th Annual International Workshop on Active Middleware Services*, June 2003.

[13] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13*, 2006.

[14] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. V. Reich. The Open Grid Services Architecture, version 1.0, Jan. 2005. http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf.

[15] A. Gurguis and A. Zeid. Towards Autonomic Web Services : Achieving Self-Healing Using Web Services. In *International Conference on Software Enginering. Workshop on Design and Evolution of Autonomic Application Software*, 2005.

[16] D. Hagimont, S. Bouchenak, N. D. Palma, and C. Taton. Autonomic Management of Clustered Applications. In *IEEE International Conference on Cluster Computing*, Barcelona, Sept. 2006.

[17] S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia: an autonomic computing environment. In *IEEE International Conference on Performance, Computing, and Communications*, Apr. 2003.

[18] G. Kaiser, P. Gross, G. Kc, J. Parekh, and G. Valetto. An Approach to Autonomizing Legacy Systems. In *Workshop on Self-Healing, Adaptive and Self-MANaged Systems, SHAMAN*, New York City, June 2002.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *9th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Austin, TX, Nov. 2000.

[20] MySQL. MySQL Web Site. http://www.mysql.com/.

[21] PLB. PLB - A free high-performance load balancer for Unix. http://plb.sunsite.dk/.

[22] C. Reich, M. Banholzer, and R. Buyya. Autonomic Container for Hosting WSRF-based Web Services. In *Technical Report, GRIDS-TR-2007-1, Grid Computing and Distributed Systems Laboratory*, Melbourne, Australia, Jan. 2007.

[23] R. Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining. *ACM Transaction on Computer Systems*, 21(2), 2003.

[24] Sun Microsystems. Java 2 Platform Enterprise Edition (J2EE). http://java.sun.com/j2ee/.

[25] Sun Microsystems. Java DataBase Connection (JDBC). http://java.sun.com/jdbc/.

[26] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.