Self-Manageable Replicated Servers

Christophe Taton[1], Sara Bouchenak[2], Fabienne Boyer[2],
Noël De Palma[3], Daniel Hagimont[1], Adrian Mos[1]

[1] INRIA
Grenoble, France

[2] University of Grenoble I
Grenoble, France

[3] INPG
Grenoble, France

{Christophe.Taton, Sara.Bouchenak, Fabienne.Boyer, Noel.Depalma, Daniel.Hagimont, Adrian.Mos}@inria.fr

Abstract

This paper describes a middleware solution for self-manageable and autonomic systems, and presents its use with replicated databases. Preliminary case studies for automatically recovering from server failures and for automatically adapting a cluster of replicated servers according to QoS requirements are presented.

1. Introduction

Replication is a well-known approach to provide service scalability and availability. Two successful applications are data replication [6], and e-business server replication [2][4][7]. The complexity of such systems makes their management extremely difficult as it involves multiple coordinated repair and tuning operations, and usually requires the manual help of operators with combined skills in database, middleware and operating system management.

In this paper, we propose a middleware-based solution for self-manageable and autonomic systems; and illustrate its use with replicated databases. The originality of the proposed approach is its *generality* on two axes. First, it may apply different reconfiguration strategies to tackle runtime changes, e.g. automatic recovery from failures, and automatic guarantee of a given quality of service (QoS). Second, the proposed approach is illustrated here with replicated databases; but we show that it may apply to other software components for providing them with self-management, e.g. web servers, or e-business application servers.

We implemented *Jade*, a prototype of the proposed middleware-based solution for self-manageable systems. We then used Jade with an e-business web application relying on databases replicated in a cluster. Our preliminary experiments illustrate the usefulness of Jade for ensuring QoS and availability requirements.

The remainder of the paper is organized as follows. Section 2 presents an overview of the Jade middleware for the management of autonomic systems. Section 3 describes scenarios in which Jade was used for ensuring QoS requirements and providing failure management. Finally, section 4 presents our conclusions and future work.

2. JADE middleware for autonomic systems

This section first introduces the main design principles of the Jade management system, before discussing QoS management and failure management in Jade.

2.1. Design principles

Jade is a middleware for the management of autonomic computing systems. Figure 1 describes the general architecture of Jade and its main features and reconfiguration mechanisms, namely the *QoS Manager* and the *Failure Manager*. Roughly speaking, each Jade's reconfiguration mechanism is based on a control loop with the following components:

– First, *sensors* that are responsible for the detection of the occurrence of particular events, such as. a database failure, or a QoS requirement violation.
– Second, *analysis/decision* components that represent the actual reconfiguration algorithm, e.g. replacing a failed database by a new one, or increasing the number of resources in a cluster of replicated databases upon high load.
– Finally, *actuators* that represent the individual mechanisms necessary to implement reconfiguration, e.g. allocation of a new node in a cluster.

Figure 1 illustrates the use of Jade with an e-business multi-tier web application distributed in a cluster, which consists of several components: a web server as a front-end, two replicated enterprise servers in the middle-tier, and four replicated database servers as a back-end.

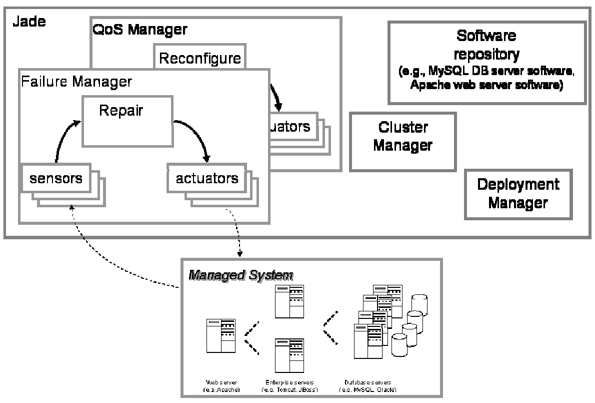
Jade
QoS Manager
Reconfigure
Failure Manager
Repair
sensors
actuators
uators
Software repository (e.g., MySQL DB server software, Apache web server software)
Cluster Manager
Deployment Manager
Managed System
Web server (e.g. Apache)
Enterprise servers (e.g. Tomcat, JBoss)
Database servers (e.g. MySQL, Oracle)

Figure 1. JADE architecture

Jade provides a *Deployment Manager* which automates and facilitates the initial deployment of the managed system. To that purpose, the *Deployment Manager* makes use of two other mechanisms in Jade: the *Cluster Manager* and the *Software Repository*.

The *Cluster Manager* is responsible for the management of the resources (i.e. nodes) of the cluster on which the managed system is deployed. A node of the cluster is initially free, and may then be used by an application component, or may have failed. The *Cluster Manager* provides an API to allocate free nodes to the managed system/release nodes after use. Once nodes are allocated to an application, Jade deploys on those nodes the necessary software components that are used by the managed system.

The *Software Resource Repository* allows the automatic retrieval of the software resources involved in the managed application. For example, in case of an e-business multi-tier J2EE [8] web application, the used software resources may be a MySQL database server software, a JBoss enterprise server software, and an Apache web server software [5].

Once nodes have been allocated by the *Cluster Manager* and software resources necessary to an application retrieved from the *Software Resource Repository*, those resources are automatically deployed on the allocated nodes. This is made possible due to the API provided by nodes managed by Jade, namely an API for remotely deploying software resources on nodes.

The Jade prototype was implemented using a Java-based and free open source implementation of a software component model called Fractal [3]. Moreover, the software resources (e.g. MySQL server software) used by the underlying managed system are themselves encapsulated in Fractal components which homogeneously exhibit management-related interfaces, such as the lifecycle interface (e.g. start/stop operations). Therefore, this helps to provide a generic implementation of the Jade management system with a uniform view of all the managed software components, regardless of whether those components actually represent different legacy software systems such as MySQL or Postgres. Abstracting the managed software as Fractal components enables the development of advanced deployment services. Moreover, the component model provides dynamic component introspection capabilities that are used for reconfiguration operations.

2.2. QoS manager

One important autonomic administration behavior we consider in Jade is self-optimization. Self-optimization is an autonomic behavior which aims at maximizing resource utilization to meet the end user needs with no human intervention required. A classical pattern in a standard QoS infrastructure is depicted by Figure 2. In such pattern, a given resource *R* is replicated statically at deployment time and a front-end proxy *P* acts as a load balancer and distributes incoming requests among the replicas.

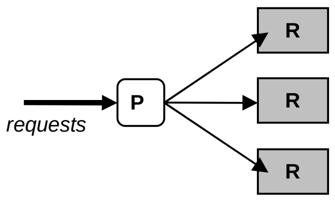

Figure 2. Load balancing among replicas

Jade aims at autonomously increasing/decreasing the number of replicated resources used by the application when the load increases/decreases. This has the effect of efficiently adapting resource utilization (i.e. preventing resource overbooking).

To this purpose, the QoS manager uses sensors to measure the load of the system. These sensors can probe the CPU usage or the response time of application-level requests. The QoS manager also uses actuators to reconfigure the system. Thanks to the generic design of Jade, the actuators used by the QoS manager are themselves generic, since increasing/decreasing the number of resources of an application is implemented as adding/removing components in the application structure.

Besides sensors and actuators, the QoS manager makes use of an analysis/decision component which is responsible for the implementation of the QoS-oriented self-optimization algorithm. This component receives notifications from sensors and, if a reconfiguration (resource increase) is required, it increases the number of resources by contacting the *Cluster Manager* to allocate available nodes. It then contacts the *Software Resource Repository* to retrieve the necessary software resources, deploys those software resources on the new nodes and adds them to the existing application structure.

Symmetrically, if the resources allocated to an application are under-utilized, the QoS manager performs a reconfiguration to remove some replicas and release their resources (i.e. nodes).

To summarize, Figure 3 describes the main operations performed by the QoS manager, which are the following:

If more resources are required:

- Allocate free nodes for the application
- Deploy the required software on the new nodes
- Perform state reconciliation with other replicas if necessary
- Integrate the new replicas to the load balancer.

If some resources are under-utilized:

- Unbind some replicas from the load balancer
- Stop those replicas
- Release the nodes hosting those replicas if no more used.

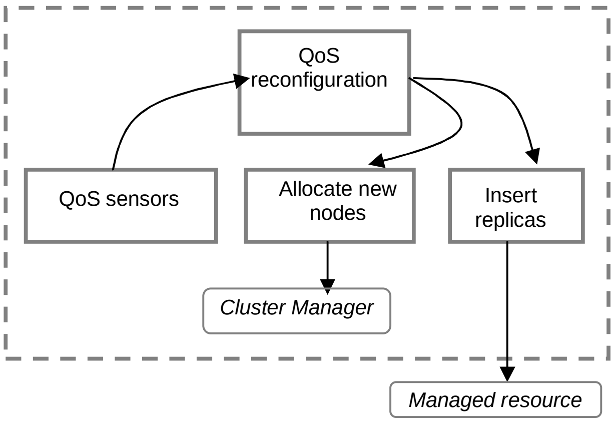

Figure 3. QoS management

2.3. Failure manager

Another autonomic administration behavior we consider in Jade is self-repair. In a replication-based system, when a replicated resource fails, the service remains available due to replication. However, we aim at autonomously repairing the managed system by replacing the failed replica by a new one. Our current goal is to deal with fail-stop faults. The proposed repair policy rebuilds the failed managed system as it was prior to the occurrence of the failure. To this purpose, the failure manager uses sensors that monitor the health of the used resources through probes installed on the nodes hosting the managed system; these probes are implemented using heartbeat techniques. The failure manager also uses a specific component called the *System Representation*. The *System Representation* component maintains a representation of the current architectural structure of the managed system, and is used for failure recovery. One could state that the underlying component model could be used to dynamically introspect the current architecture of the managed system, and use that structure information to recover from failures. But if a node hosting a replica crashes, the component encapsulating that replica is lost; that is why a *System Representation* which maintains a backup of the component architecture is necessary. This representation reflects the current architectural structure of the system (which may evolve); and is reliable in the sense that it is itself replicated to tolerate faults. The *System Representation* is implemented as a snapshot of the whole component architecture.

Besides the system representation, the sensors and the actuators, the failure manager uses an analysis/decision component which implements the autonomic repair behavior. It receives notifications from the heartbeat sensors and, upon a node failure, makes use of the *System Representation* to retrieve the necessary information about the failed node (i.e., software resources that were running on that node prior to the failure and their bindings to other resources). It then contacts the *Cluster Manager* to allocate a new available node, contacts the *Software Resource Repository* to retrieve the necessary software resources and redeploys those software resources on the new node.

The *System Representation* is then updated according to this new configuration. Figure 4 summarizes the operations performed by the failure manager.

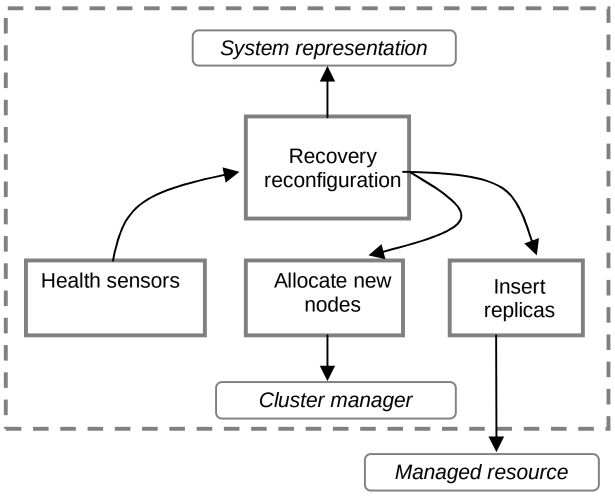

Figure 4. Failure management

Note that the same abstractions (components) and the same actuators are used to reconfigure the managed system for the QoS aspect and the failure management aspect. However, the sensors differ in these two cases. Furthermore, owing to the component abstraction and reconfiguration capabilities, this repair policy can be used to repair the management system itself, i.e. Jade, which is a Fractal-based implementation, and therefore benefits from reconfiguration capabilities of that software component model.

3. Case Studies

In order to validate our management approach, we have implemented and tested several use-cases related to QoS and failure management.

Our first experiments involved stateless replicated servers (i.e. Apache web server and Tomcat application server). In addition we implemented a stateful read-only case-study and we are working on adding read-write support for replica recovery.

All the experiments used the Rubis benchmark [1] as the application environment. Rubis is an auction application prototype similar to eBay and intended as a performance benchmark for application servers. It is therefore appropriate for the validation of cluster management functionality present in Jade.

3.1. QoS management experiments

Stateless replicas

This experiment involves dynamic resizing of a cluster of Apache web-servers delivering static pages. All servers (active and idle) contained identical content and activating one server implied using Jade's dynamic deployment to deploy Apache on an idle node.

The web load is distributed by a proxy P to the replicated Apache (A) servers. Figure 5 illustrates that an active node can be automatically removed or an idle node can be automatically added, based on workload variations. The QoS sensor in this case is monitoring the workload received by P.

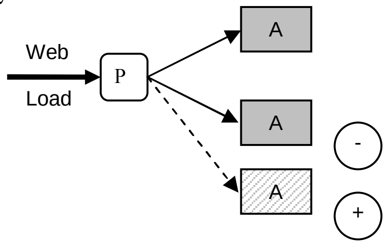

Figure 5. QoS management of stateless replicas

Stateful replicas / read-only access

Dynamic cluster-resizing, illustrated in Figure 6, is applied in this experiment to a set of DB replicas serving a read-only client load. As an optimization, we preloaded the same database content on all nodes (active and idle). In our experiments, the load-balancer among replicated databases is c-jdbc [6].

The database load, arriving from the web server is distributed by c-jdbc to the DB replicas that can be added and removed based on workload variations. The QoS sensor in this case is monitoring the workload received by the c-jdbc controller.

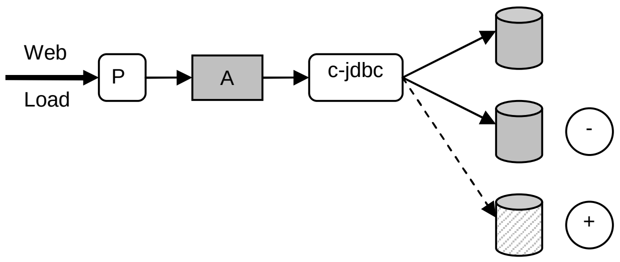

Figure 6. QoS management of R/O stateful replicas

Stateful replicas / read-write access

We are currently working on providing the same functionality in scenarios with read-write client loads. The technique we use leverages the logging facilities of c-jdbc. For each node activation, the manager will perform the deployment operation on the node, thus bringing it to the initial state of all the database nodes (as in the previous use-case, all nodes have the DB state preloaded). In order to update the new node so as to synchronize it with the other replicas, the log file is used to replay all the SQL statements that have been recorded since the last state synchronization. Figure 7 illustrates the reconciliation operation performed as part of node activation. This is a relatively fast operation; however it depends on the time between state synchronizations and the number of writes during this time.

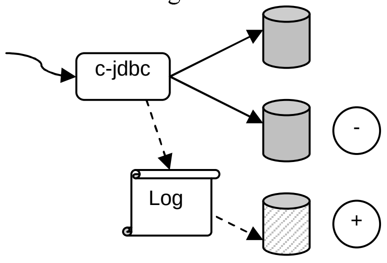

Figure 7. Reconciliation of a new R/W replica

3.2. Failure management experiments

We have tested Jade's ability to repair running systems in a Rubis web application scenario in which we had a cluster of 4 Tomcat servers serving dynamic content to one Apache server. The Tomcat servers were connected to a MySQL database holding the application data.

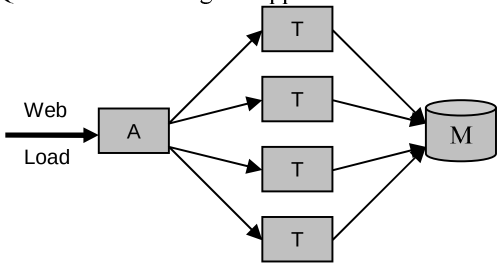

Figure 8. Failure management case study

The case-study's architecture is illustrated in Figure 8.

We induced 3 consecutive Tomcat server crashes in order to observe the evolution of the application performance when not being managed by Jade as well as when under Jade's management.

When the system was not under Jade management, the only remaining server saturated and the response time perceived by the client emulator increased dramatically, essentially rendering the system unavailable.

When the system was managed by Jade, the failure manager automatically recovered the crashed servers. This demonstrates Jade's capacity to dynamically repair the affected parts of the software architecture and preserve system availability. Note that this assumes that either a pool of available nodes exists, or that the cause of the crashes is a software malfunction and the same nodes can be reused after a restart and redeployment.

The presented experiments involved stateless replicas, i.e. replicas whose internal state did not need to be preserve between crashes. We plan to perform the same experiments in a scenario with replicated databases. As such, we would induce consecutive DB server crashes and observe the repair functionality of Jade involving state reconciliation operations (see section 3.1).

4. Conclusion and future work

Managing replicated systems is a complex task, in particular in large enterprise settings that deal with important variations in resource utilization and server crashes. We presented a middleware solution that enables automatic reconfiguration and repair of large clusters of database, web and application servers, thus limiting the need for costly and slow manual interventions.

By encapsulating all architectural entities in a consistent component model, Jade provides a uniform management framework capable of enforcing QoS and availability constraints in heterogeneous deployments.

We demonstrated the QoS management operations with experiments that involved automatic resizing of web and database clusters for preserving optimal resource utilization. In addition we illustrated the failure recovery functionality by contrasting the evolution of a system with and without Jade management, in a replicated enterprise environment with induced failures.

For future work we will consolidate the implementation of the QoS and failure managers to better deal with read-write scenarios in DB clusters. The general aspect of Jade's management approach will allow us to provide a consistent set of operations valid for all DB servers, while at the same time have efficient state reconciliation techniques that leverage DB-specific optimizations.

5. References

[1] 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. *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, USA, Nov. 2002. http://rubis.objectweb.org

[2] BEA WebLogic. Achieving Scalability and High Availability for E-Business, January 2004. http://dev2dev.bea.com/pub/a/2004/01/WLS_81_Clustering.html

[3] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and Dynamic Software Composition with Sharing. *7th International Workshop on Component-Oriented Programming (WCOP02)*, Malaga, Spain, June 10, 2002. http://fractal.objectweb.org/

[4] B. Burke, S. Labourey. Clustering With JBoss 3.0. October 2002. http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html

[5] R. Cattell, J. Inscore. J2EE Technology in Practice: Building Business Applications with the Java 2 Platform, Enterprise Edition. *Pearson Education*, 2001.

[6] E. Cecchet, J. Marguerite, W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. *FREENIX Technical Sessions, USENIX Annual Technical Conference*, Boston, MA, Etats-Unis, June 2004. http://c-jdbc.objectweb.org/

[7] G. Shachor. Tomcat Documentation. The Apache Jakarta Project. http://jakarta.apache.org/tomcat/tomcat-3.3-doc/

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