Abstract

Nowadays, medium or large-scale distributed infrastructures such as clusters and grids are widely used to host various kinds of applications (e.g. web servers or scientific applications). Power consumption is becoming a major challenge for most organizations that run these infrastructures. Many studies show that they are not used at their full capacity and that there are therefore a huge source of wasted power.

Autonomic management systems have been introduced in order to dynamically adapt software infrastructures according to runtime conditions. They provide support to deploy, configure, monitor, and repair applications in such environments.

In this paper, we report our experiments in using Tune - an autonomic management system - to provide energy aware management for a clustered J2EE application. We use Tune to dynamically adapt the degree of replication of the J2EE tiers in the cluster, and to dynamically turn cluster nodes on - to handle the load when it raises up - and off - to save power under lighter load.

1. Introduction

Nowadays, medium or large-scale distributed infrastructures such as clusters and grids are widely used to host various kinds of applications (e.g. web servers or scientific applications). The development of Cloud Computing[6] illustrates a general trend towards the emergence of large-scale hosting centers.

Power consumption is becoming a major challenge for most organizations that run these infrastructures. According to the U.S. Environmental Protection Agency (EPA) [9], it is estimated that this sector consumed about 61 billion kilowatt-hours (kWh) in 2006 (1.5 percent of total U.S. electricity consumption) for a total electricity cost of about $4.5 billion. Moreover, energy consumption of these infrastructures is estimated to have doubled between 2000 and 2006 and the development of hosting centers will amplify this increase.

Studies [14, 18] made on such infrastructures show that they are not used at their full capacity. As there is a relatively little difference in power consumption between an idle node and a fully used node, the penalty (regarding energy) for keeping an idle node powered on is high. Moreover, such hosting infrastructures require large cooling systems which also consume a lot.

Autonomic management systems [15] have been proposed as one solution for the management of distributed infrastructures. Such systems can be used to deploy and configure applications in a distributed environment. They can also monitor the environment and react to events such as failures or overloads and reconfigure applications accordingly and autonomously.

Tune [4] is such an autonomic management system that we designed and implemented. It provides a high level formalism for the specification of all administration tasks (wrapping, deployment, reconfiguration). In Tune, an administrator can wrap legacy software in components, describe a software environment to deploy, and implement reconfiguration programs according to its needs. We have shown the benefits of our approach with several application examples.

In this context, we aim at using Tune in order to dynamically adapt the architecture of an application according to the load it receives. This adaptation is a means to dynamically allocate or free machines, i.e. to dynamically turn cluster nodes on - to be able to efficiently handle the load imposed on the system - and off - to save power under lighter load. We implemented such a management policy for a clustered J2EE application and evaluated the benefits for energy consumption.

The rest of paper is structured as follows: Section 2 presents the context of our work and our motivations. Section 3 describes our approach. Section 4 presents the experiments to evaluate our approach. After an overview of related works in Section 5, we conclude in Section 6.
2. Context

In this section, we first present the application case that we use to illustrate our approach. We then present in more details the Tune autonomic management system that we used in our experiments.

2.1 Clustered J2EE Application

As experimental environment, we made use of the Java 2 Platform, Enterprise Edition (J2EE), which defines a model for developing web applications [17] in a multi-tiered architecture. Such applications are typically composed of a web server (e.g. Apache), an application server (e.g. Tomcat) and a database server (e.g. Mysql). Upon an HTTP client request, either the request targets a static web document, in this case the web server directly returns that document to the client; or it refers to a dynamically generated document, in that case the web server forwards the 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). 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 for highly scalable and highly available services. To deal with high loads and provide higher scalability of Internet services, a commonly used approach is the replication of servers in clusters. Such an approach (Figure 1) usually defines a particular software component in front of each set of replicated servers, which dynamically balances the load among the replicas. Here, different load balancing algorithms may be used, e.g. Random, Round-Robin, etc.

In such an architecture, a difficult issue is the find the best degree of replication for each tier, which should be sufficient to tolerate load peaks, but not too high to prevent resource wasting.

2.2. Toulouse University Network: TUNe

Many works in this area [11, 12] have relied on a component model to provide such an autonomic system support. The basic idea is to encapsulate the managed elements (legacy software) in software components and to administrate the environment as a component architecture. Consequently, the administrators can benefit from the essential features of the component model, encapsulation, deployment facilities and reconfiguration interfaces, in order to implement their autonomic management processes.

Tune [4] is a component-based autonomic management system which is developed in the IRIT1 laboratory. Tune relies on the Fractal component model [5]. Each software administrated by Tune is encapsulated (wrapped) in a Fractal component and the software environment managed as a component architecture, but in order to allow the administration by non-experts, Tune provides higher level formalisms for the description of the software architecture, its deployment and reconfiguration policies. These formalisms take the form of UML profiles2. We detail these formalisms in the next sub-sections.

2.2.1 A UML profile for deployment schemas

Tune introduces a UML profile for graphically describing deployment schemas. A deployment schema describes the overall organization of a software infrastructure to be deployed. At deployment time, the schema is interpreted to deploy component architecture. The UML profile illustrated in Figure 2 represents a deployment schema defined for a J2EE application.

Each element (the boxes) corresponds to a software which can be instantiated in several component replicas. A link between two elements generates bindings between the components instantiated from these elements. An element includes a set of configuration attributes for the software. Most of these attributes are specific to the software, but few attributes are predefined by Tune and used for deployment.

Figure 2’s deployment schema includes a frontend load-balancer (HAProxy), a Tomcat server, a database load-balancer (Cjdbc) and a database server (Mysql). Tomcat and Mysql servers are monitored by probes. In this deployment schema, each component may be replicated by adapting the initial attribute, which indicates the number of instances that have to be deployed.

2.2.2 A Wrapping Description Language

Upon deployment, the above schema is parsed and for each element, a number of Fractal components are created.

---

1Institut de Recherche en Informatique de Toulouse
2Unified Modeling Language
These Fractal components implement the wrappers for the deployed software, which provide control over the software. Each wrapper Fractal component is an instance of a generic wrapper which is actually an interpreter of a WDL specification.

A WDL description defines a set of methods that can be invoked to configure or reconfigure the wrapped software. The workflow of methods that have to be invoked in order to configure and reconfigure the overall software environment is defined thanks to a formalism introduced in Section 2.3. Generally, a WDL specification provides `start` and `stop` operations for controlling the activity of the software, and a `configure` operation for reflecting the values of the attributes (defined in the UML deployment schema) in the configuration files of the software. Notice that the values of these attributes can be modified dynamically. Other operations can be defined according to the specific management requirements of the wrapped software, these methods being implemented in Java.

Figure 3 shows an example of WDL specification which wraps a Tomcat server in a J2EE architecture. It defines `start` and `stop` methods which can be invoked to launch/stop the deployed Tomcat software, and a `configure` method which reflects configuration attributes in the configuration file of the Tomcat software. The Java implementations of these methods are generic and have been used in the wrappers of most of the software we wrapped. A method definition includes the description of the parameters that should be passed when the method is invoked. These parameters may be String constants, attribute values or combination of both (String expressions). All the attributes defined in the deployment schema can be used to pass the configured attributes as parameters of the method invocations.

### 2.2.3 A UML profile for (re)configuration procedures

Tune introduces a UML profile for the description of state diagrams. These state diagrams are used to define workflows of operations that have to be performed for reconfiguring the managed environment. Reconfigurations are triggered by events. An event can be generated by a specific monitoring component (e.g. probes in the deployment schema) or by a wrapped legacy software which already includes its own monitoring functions.

Whenever a wrapper component is instantiated, a communication pipe is created (typically a Unix pipe) that can be used by the wrapped legacy software to generate an event, following a specified syntax which allows for parameter passing. Notice that the use of pipes allows any software to generate events. An event generated in the pipe associated with the wrapper is transmitted to the administration node where it can trigger the execution of reconfiguration programs (in our current prototype, the administration code, which initiates deployment and reconfiguration, is executed on one administration node, while the administrated software is managed on distributed hosts). An event is defined as an event type, the name of the component which generated the event and eventually an argument (all of type String).

For the definition of reactions to events, we introduced a UML profile which allows specifying reconfiguration as state diagrams. Such a state diagram defines the workflow of operations that must be applied in reaction to an event.

---

**Figure 3. A WDL specification**
An operation in a state diagram can assign an attribute or a set of attributes of components, or invokes a method or a set of methods of components.

For example, let’s consider the diagram in Figure 4 which is the diagram for adding a new Mysql server. The event (upsizemySql) is generated by a ProbeMysql component instance, therefore the this variable is the name of this MysqlProbe component instance. Then:

- **this.stop** will invoke the stop method on the probing component (to prevent the generation of multiple events),
- **new=this.Mysql++** will create a new Mysql instance. Notice here that in order to designate the component on which an operation should be performed, the syntax allows navigation in the component architecture, e.g. here to follow the link between the ProbeMysql and the Mysql elements. The ++ operator increases the number of instances of the Mysql element.
- **new.start** will invoke the start method on the created Mysql component instance.
- **Cjdbc.configure** will invoke the configure on the database load-balancer to take into account the created Mysql component instance.
- **Cjdbc.stop** will stop the database load-balancer.
- **Cjdbc.start** will restart the database load-balancer to take into account the created Mysql.
- **this.start** will restart the probe associated with the Mysql element.

### 2.3. Motivations

Tune has proven to be a very convenient autonomic management system. The experiments we conducted with the management of real applications validated its design choices. We have shown that Tune can be used for automatically deploying, configuring, monitoring, repairing and dimensioning applications in a distributed environment [4].

But in our previous experiments, we did not address the issue of energy management, which is a very interesting application for an autonomic management framework like Tune. Based on the fact (Section 1) that infrastructures like clusters and grids are not used at their full capacity, it should be very valuable to implement with Tune a self-sizing autonomic policy which dynamically adjusts the size of the application architecture as needed. Such a policy would aim at:

- Guaranteeing the performance of the application by adapting its size as needed.

### 3. Autonomic Energy Management For Clustered J2EE Architecture

In this section, we present the management policy that we implemented with Tune in order to manage energy in a J2EE cluster.

#### 3.1. self-sizeable clustered application

A standard pattern for implementing scalable clustered servers is the load balancer. In this pattern, a given application server (e.g. Tomcat in Figure 2) is replicated and a frontal proxy (e.g. HAProxy) distributes incoming requests among the replicated servers. We have implemented with Tune an autonomic policy which adjusts the number of replicas used by the application when the load varies. This implementation relies on probes (described in Figure 2) which monitor the load of the replicated servers.

The deployment diagram in Figure 2 is parameterized by different factors which control the addition and removal of replicas:

- **maxCPU**: this parameter specifies the maximum cpu
load that a node can reach before triggering the addition of a replica.

- **minCPU**: this parameter specifies the minimum CPU load a machine can reach before triggering the removal of a replica.

- **maxServerCount**: this parameter specifies the maximum number of replica.

The J2EE architecture is initially deployed with one application server (Tomcat) and one database server (Mysql), as described in Figure 5. We use two probes: one for Tomcat servers and the other for Mysql servers. These probes periodically collect the CPU usage information on all the nodes where Tomcat and Mysql servers are deployed. They compute a moving average of the collected data in order to remove artifacts characterizing the CPU consumption. They finally compute an average CPU load across all nodes, so as to observe a general load indication of the whole replicated server. From this average, the probe will eventually generate one of the following events:

- **Adding a new replica**: Tune allocates a node from a list of available nodes for this server. Tune turns the allocated node on if it is turned-off, and it applies the reconfiguration diagram corresponding to the addition of new server when the node is ready, as described in Figure 4. The J2EE architecture after this operation is illustrated in Figure 6.

- **Removing a replica**: In this case, Tune applies the reconfiguration diagram which removes a replica and frees the node.

3.2. Suspension to RAM

Turning machines off and (especially) on is quite costly. We measured that such an operation costs about 45 seconds in the average. Instead, we rely on suspension to RAM, which allows to suspend and resume the activity of a machine at a low cost (about 4 seconds in the average for resuming a machine) while saving as much energy as if it was turned off. Suspend-to-RAM stores information on system configuration, open applications, and active files in main memory (RAM), while most of the system’s other components are turned off. When a machine is suspended, only the RAM and the network device are power on.

4. Evaluation

In this section, we present the evaluation of our solution.

**Testbed application**

The evaluation has been realized with RUBiS [3], a J2EE application benchmark based on servlets, which implements an auction site modeled over eBay. It defines 26 web interactions, such as registering new users, browsing, buying or selling items. RUBiS also provides a benchmarking tool that emulates web client behaviors and generates a tunable workload. This benchmarking tool gathers statistics about the generated workload and the web application behavior.

**Hardware environment**

The experimental evaluation has been performed on a cluster of x86-compatible machines. The experiments required up to 7 machines: one node for Tune management platform and HAProxy, one node for Tomcat, one node for Cjdbc, up to three nodes for Mysql, one node for RUBiS client emulator (which emulates up to 1000 clients). The number of nodes actually used during these experiments varies, according to the dynamic changes of the workload, and thus to the dynamic resizing of the application. All the nodes are connected through a 100Mbps Ethernet LAN to form a cluster. For energy measurement, we used a programmable power meter (HAMEG HM8115-2).

**Software environment**

The nodes run version 2.6.30 of the Linux kernel. The J2EE application has been deployed using open source
middleware solutions: Jakarta Tomcat 6.0.20 [10] for the web and servlet servers, Mysql 5.1.36 [2] for the database servers, Cjdbc 2.0.2 [7] for the database load-balancer, HAPerxy 1.3 [1] for the application server load-balancer. We used RUBiS 1.4.3 as the running J2EE application. These experiments have been realized with Sun’s JVM JDK 1.6.0.05. We used the Mysql Connector/J 5.1 JDBC driver to connect the database load-balancer to the database servers.

4.1. Evaluation scenario

We aim at showing that dynamic allocation and deallocation of nodes in response to workload variations allows energy saving with tolerable performance degradation. In this evaluation, we provide measurements for the database replicated tier only.

In our benchmark, the load increases progressively up to 1000 emulated clients: 50 new emulated clients every minute. Since a RUBiS client runs for 25 minutes, the total time of the experiment is about 45 minutes.

During the experiment, the probe monitors the CPU usage of Mysql server nodes. If the CPU usage gets higher than the value given by the maxCPU parameter, the probe sends a notification to Tune asking to add a new server. If the CPU usage gets lower than the minCPU parameter, the probe sends a notification to Tune asking to remove the Mysql server. The J2EE architecture is initially deployed with one database server (Mysql) and Tune reacts to the load variation by allocating and freeing nodes, as described in section 3.1.

We compare energy consumption and quality of service in two situations:

- static configuration of the J2EE architecture. We measure the performance (regarding quality of service and energy consumption) with one, two and three database servers. Fewer servers will save energy but with a degradation of quality of service. More servers will optimize quality of service, but with energy wasting.

- dynamic configuration of the J2EE architecture. Tune is used to dynamically adapt the number of database servers as described above. Therefore, quality of service is guaranteed without wasting energy.

4.2. Result

In a first evaluation, we measure the throughput and the response time of the J2EE application when statically configured with one, two and three Mysql servers. These measurements are given in Figure 7 and Figure 8.

In Figure 7, we observe that a single Mysql server can maintain the quality of service for up to 450 clients, and up to 900 clients for two database servers. The configuration with three database servers can maintain the quality of service for all the emulated clients. We observe the same effects in Figure 8 regarding response time.

We also observed that with Tune, throughput and the response time are the same as with three database servers, as Tune adapts the number of servers according to the load (more details are given in Figure 11).

In conclusion, in this experiment, maintaining the quality of service requires three database servers, but these three servers are not required during the whole experiment, which leads to energy wasting that we evaluate in the following.

Figure 9 gives energy consumption of the J2EE application when statically configured with one, two and three Mysql servers. Obviously, the more machines you use, the more you consume.

With Tune, we dynamically turn cluster nodes on - to be able to handle load peaks - and off - to save power under lighter load. We only use nodes (and consume energy) when needed. Figure 10 shows the consumed energy when using Tune. We start with only one Mysql server, when this server approaches the saturation point, Tune adds a new server to the architecture. This occurs twice at times 600 and 1000.

During this experiment, if we compare the energy consumption of the static configuration with 3 database servers, with the energy consumption with Tune (both being able to
maintain quality of service), we observe that Tune reduces energy consumption by 21%.

Another evaluation metric is the number of requests handled by the web server during the experiment. Indeed, when the server saturates, some requests cannot be handled in time and are therefore rejected. Figure 11 gives the number of handled requests during this experiment for the different considered configurations. The left bar gives the consumed energy and the right bar the number of handled requests. We observe that Tune is close to the optimum regarding handled requests and that it provides the best tradeoff between quality of service and energy.

5. Related work

Many works have addressed the issue of power management. Most of them have focussed on energy management for electronic devices powered by electric battery, and few have addressed this issue in grid or cluster infrastructures.

Many research works which aim at managing energy for a single processor system can be used to optimize energy consumption independently on each node of a cluster. This can include optimizations of the use of the processor, memory, disk, etc...

Most of the research that focusses on cluster-wide energy management deals with resource allocation. We can mention some examples of such works:

- **Load balancing** In this category, we can cite the work of E. Pinheiro et al. [19] who developed an algorithm which makes load balancing decisions by considering both the total load imposed on the cluster and the power and performance implications of turning nodes off.

- **Virtualization** In this category, we can cite the work of Hermenier et al. [13] who developed a system which relies on virtual machine migration for transparently relocating any application in the cluster. The placement policy takes into account the cpu and memory usages, in order to concentrate the workload on fewer nodes of the cluster, thus allowing unused nodes to be shutdown. We are currently cooperating with them to integrate our autonomic managment system with their work.

- **Simulation** We can here cite the work of Khargharia et al. [16]. They present a theoretical framework and methodology for autonomic power and performance management in data centers. They rely on simulation to apply their approach to two case studies, a multi-chip memory system and a high performance server cluster.

Our work is orthogonal to these contributions. While most of the works made in this domain is specific to energy management, our autonomic computing approach is generic, as it can be used to define any management policy for any distributed software architecture. The field of energy management was not previously addressed by Tune, but the experiment reported in this paper shows that Tune can by used to define an energy management autonomic policy.
The closest work to our is that of Kephart et al. [8] who proposed a multi-agent approach for managing power and performance in server clusters by turning off servers under low-load conditions. Instead of relying on components and architectures, their autonomic system follows a multi-agent paradigm.

6. Conclusion and Future Work

Nowadays, medium or large-scale distributed infrastructures such as clusters and grids are widely used to host various kinds of applications. Power consumption has become a major challenge for most organizations that run these infrastructures. Many studies show that they are not used at their full capacity and that there are therefore a huge source of wasted power. Autonomic management systems have been recognized as a convenient solution for management of distributed infrastructures.

The experiments that we conducted show that the autonomic computing approach can be used for energy management in a distributed infrastructure. This approach meets the need of energy aware computing, which can be summarized in: minimize power consumption, without affecting the performance of the system. In our experiments, we were able to reduce the power consumption of a typical web application benchmark by approximately 21%.

This paper reported on preliminary work. In the near future, we aim at evaluating much deeply our prototype through more elaborated power management policies, which would include other parameters, for example, network traffic information. We also wish to integrate virtualization techniques in our prototype, as it would enable transparent process (VM) migration between hardware nodes.

7  Acknowledgments

The work reported in this paper benefited from the support of the French National Research Agency through project SelfXL (ANR-08-SEGI-017-04).

References