

VINICIUS TAVARES PETRUCCI

**A framework for supporting dynamic adaptation of
power-aware web server clusters**

NITERÓI

2008

VINICIUS TAVARES PETRUCCI

**A framework for supporting dynamic adaptation of
power-aware web server clusters**

Dissertation submitted to the Graduate School of Computation of Fluminense Federal University as a partial requirement for the degree of Master in Science.

Topic Area: Parallel and Distributed Computing.

Supervisor:

Prof. Orlando Gomes Loques Filho, Ph.D.

UNIVERSIDADE FEDERAL FLUMINENSE

NITERÓI

2008

A framework for supporting dynamic adaptation of power-aware web server clusters

Vinicius Tavares Petrucci

Dissertation submitted to the Graduate School of Computation of Fluminense Federal University as a partial requirement for the degree of Master in Science.

Topic Area: Parallel and Distributed Computing.

Approved by:

Prof. Orlando Gomes Loques Filho, Ph.D. (IC/UFF)
(Chair)

Prof. Célio Vinicius Neves de Albuquerque, Ph.D. (IC/UFF)

Prof. Claudio Luis de Amorim, Ph.D. (COPPE/UFRJ)

Niterói, December 2008.

If you want to make an apple pie from scratch, you must first create the universe.

— Carl Sagan (1934 - 1996)

Abstract

Modern software applications are increasingly being deployed in highly dynamic computing environments, leading to frequent changes in their execution environment, such as workload variation, changing resource availability, and component faults. In order to effectively cope with dynamic scenarios and take advantage of available resources, applications need to be able to adapt their configuration dynamically at run-time in response to changes in their execution environment, preferably without requiring any external intervention from administrators and developers. For instance, web applications in a cluster experience large periods of low utilization and present an opportunity for using dynamic adaptation techniques to reduce energy consumption with little impact on performance and timeliness properties.

In recent years, many adaptive policies for web server cluster power management have been investigated. The development of these adaptive policies may be complex in itself, and the required support mechanisms for these policies are implemented in an ad-hoc fashion, or by means of low level (built-in) operating system modules. As a result, the adaptation logic and mechanisms are hard coded (and mixed) within the application code, making it difficult to reuse the basic implementation to experiment with different adaptation requirements. In the medium term, when the application goes to a real operational environment, using these ad-hoc approaches substantially hinders application maintenance and evolution activities.

In this work, we present a framework-based approach for dynamic adaptation of distributed applications. The proposed framework is used to introduce dynamic adaptation capabilities, intended to address power and performance management, into a server cluster infrastructure. Our approach consists of providing a reusable infrastructure with common elements to monitor and adapt running applications, and an adaptation language to enable one to express high-level adaptation policies, both designed separately from the target application. By using external adaptation mechanisms, our approach enables one to modify and reason about application's adaptation logic with ease. In addition, it allows to reuse the adaptation infrastructure across different adaptation requirements, helping to reduce the cost of engineering such adaptive applications.

Keywords: dynamic adaptation, autonomic computing, frameworks, software architecture, server clusters, power management.

Contents

1	Introduction	p. 7
2	Related work	p. 11
2.1	CR-RIO	p. 11
2.2	CASA	p. 12
2.3	Rainbow	p. 13
2.4	JADE	p. 14
2.5	IBM autonomic systems	p. 15
2.6	Summary	p. 16
3	The framework	p. 17
3.1	Software architecture model	p. 17
3.2	Contract-based adaptation language	p. 18
3.2.1	Utility-based adaptation negotiation	p. 21
3.3	Adaptation support infrastructure	p. 22
3.3.1	Supporting multiple contracts	p. 24
3.4	Implementation details	p. 25
3.5	Summary	p. 26
4	Application case	p. 27
4.1	Architecture description	p. 27
4.2	Adaptation contracts	p. 28
4.2.1	Simple power management contract	p. 29

4.2.2	Decision-based power management contract	p. 32
4.2.3	Fault-tolerance contract	p. 35
4.3	Reusability and flexibility	p. 36
4.3.1	Contracts using different quality metrics	p. 37
4.4	The application-specific layer	p. 39
4.5	Summary	p. 40
5	Experimental evaluation	p. 41
5.1	Server cluster setup	p. 41
5.2	Implementation issues	p. 43
5.3	Workload generation	p. 44
5.4	Effectiveness and flexibility	p. 44
5.4.1	A comparison between adaptation policies	p. 45
5.4.2	Using different cluster quality metric	p. 46
5.5	Composing multiple contracts	p. 47
5.6	Opportunity for anticipatory adaptation	p. 48
5.7	Adaptation timing analysis	p. 49
5.8	Summary	p. 51
6	Conclusion	p. 52
6.1	Contributions	p. 52
6.2	Future work	p. 52
	References	p. 55

1 Introduction

Many of today's applications execute in highly dynamic computing environments. This leads to constant changes in their execution context, such as workload variation, changing resource availability (including component faults). In cluster-based applications, to anticipate load peaks or failures, hardware resources (e.g., servers) are often over-dimensioned, meaning they are statically allocated for providing much higher processing capacity than needed to run the application services [24, 59]. We use the term *cluster* to refer to a group of coupled computers — commonly connected to each other through a fast local area network — that work together and can be viewed as a single computer. Clusters are usually deployed to improve performance and/or availability over that provided by a single computer, while typically being much more cost-effective than single computers of comparable speed or availability [11].

A common way to achieve high availability and desirable quality-of-service is to maintain a set of spare servers to assume control in case another server fails or to start processing requests in parallel to deal with an increasing workload. The problem is that idle servers (and over-provisioned performance capacity) means additional power and cooling costs. Over the last years, server power costs have more than doubled [33]; thus, every watt saved provides real cost savings. To effectively cope with dynamic scenarios and take advantage of available resources, applications need to be able to adapt their configuration dynamically in response to changes in their execution environment, preferably without requiring any external intervention from administrators and developers.

For example, web applications in a cluster experience large periods of low utilization and present an opportunity for using dynamic adaptation techniques to reduce energy consumption with little impact on performance and timeliness [59]. That is, during low utilization some components may be completely idle, and thus could be turned off, and other components could be operated at reduced power [7, 43]. One main motivation for using dynamic adaptation on server clusters is that power and energy accounts for large fraction of their operating cost [7]. It means that power-efficiency is a very important

requirement to make server clusters even more attractive for today’s applications. Basically, power management for cluster-based systems can be divided into two categories: *inter-node* and *intra-node* [24]. In general, *intra-node* techniques aim at reducing energy consumption at CPU level, using CPU voltage/frequency scaling (DVFS), and *inter-node* techniques try to save energy on a cluster level by turning on and off machines, while complying with additional quality-of-service requirements of the web application, such as utilization or requests response time.

In recent years, many adaptive policies for cluster power management have been intensively investigated [4, 5, 8, 24, 29, 53, 59, 65]. The development of these adaptive policies may be complex in itself, and the required support mechanisms for these policies are implemented in an ad-hoc fashion [4, 59, 65], or by means of low level (built-in) operating system modules (e.g., the Linux On-Demand CPU Governor). While these approaches may give to the developer complete control over how adaptations are accomplished, they usually imply in high development costs. This happens because the adaptation logic and mechanisms are hard coded (and mixed) within the application code, making difficult to reuse the basic implementation to experiment with different adaptation requirements. In the medium term, when the application goes to a real operational environment, using these ad-hoc approaches substantially hinders application maintenance and evolution activities. According to [34], energy management is intrinsically about trading power vs. performance, availability, and it is at the middleware/framework level that such adaptation requirements (or SLAs — Service Level Agreement) should be expressed and managed.

A recent branch of work adopts the feedback control loop (also termed *autonomic computing*) paradigm as a basis to provide an engineering approach to dynamic adaptation of applications [9, 15, 26, 32, 41, 46]. The key idea is to provide a run-time, external closed loop infrastructure that monitors and collects system quality measures, processes and evaluates them, executes a procedure to select the new configuration and imposes the chosen configuration over the application (see Figure 1). The control loop approach shows four essential phases of dynamic adaptation: *monitor*, *analyze* (or detection), *plan* (or decision), and *execute* (or action). The *knowledge* element is shared between these four phases and contains models, data, and scripts — the adaptation concerns which are externalized from the managed application [16, 35].

In particular, it is recognized that framework-based approaches [16, 41, 46, 60] provide an important enabling technology (as a basic software infrastructure) to address the

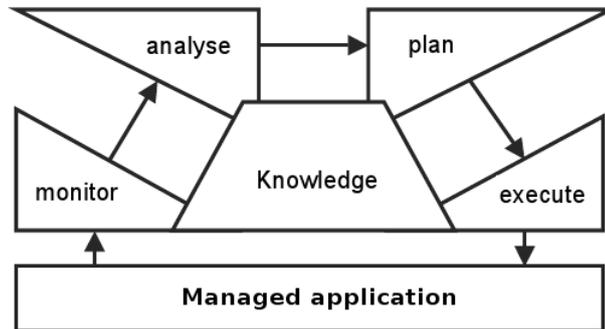


Figure 1: Closed-loop control paradigm [32].

complex engineering issues in supporting dynamic adaptations of applications. In this context, frameworks are designed with the intent of facilitating software development, by allowing designers and developers to spend more time on meeting software adaptation requirements rather than dealing with the low level details of providing the machinery to support the software adaptation mechanisms. The primary goal is to develop and implement software applications that can adapt themselves in accordance with high-level guidance from administrators and developers.

In this work, we present a framework-based solution for dynamic adaptation of distributed applications. The proposed framework is used to introduce dynamic adaptation capabilities into a web server cluster infrastructure, intended to address power and performance management, and fault tolerance concerns. Our approach consists of a reusable infrastructure with common elements to monitor and adapt running applications, and an adaptation language to enable one to express high-level adaptation policies, both designed separately from the target application. By using external adaptation mechanisms, our approach enables one to modify and reason about application’s adaptation logic with ease. In addition, it allows to reuse the adaptation infrastructure across different adaptation requirements, helping to reduce the cost of engineering new adaptive applications [16].

The proposed framework is intended to support dynamic adaptation in the development process, through the use of high-level adaptation language, as well as in the operational phase by providing a reusable adaptation infrastructure. The approach targets modularity, extensibility, and portability in providing adaptation capabilities for applications, while presenting opportunities for systematic reuse. By enabling reuse of significant parts of the adaptation infrastructure, besides facilitating the evaluation of different adaptation policies, our framework demonstrates improvement in terms of reducing the effort required for developing new adaptive applications. Also, it may help to

improve the quality of adaptive applications comparing with those developed manually from scratch.

By adopting a high-level adaptation language, our approach allows developers to abstract away complexities from low-level details of dynamic adaptation mechanisms and to concentrate on the actual adaptation logic (or policy) to be elaborated. The adaptation language further enables the adaptation policies to be changed at run-time, which is suitable for customizing it for personal user's needs and preferences, as well as for evolving the adaptation policy, e.g., for dealing with requirements unforeseen at the time of the application development, or to add new adaptation capabilities to the application. Our approach also provides support for multi-objective adaptations by composing multiple adaptation contracts.

Although our framework is targeted at web server clusters, we believe that it could be used in other infrastructures. For instance, one could use it to enable dynamic adaptation capabilities for video/audio on-demand applications in an overlay network infrastructure, as the application case shown in [68]. However, applying our framework to different kinds of adaptation contexts is outside of the scope of this work and requires further study.

The remainder of this dissertation is organized as follows: In Chapter 2 we present some background information and a brief perspective of related works to our approach. Chapter 3 covers the main elements of our proposal; specifically, the adaptation infrastructure and the adaptation language. In Chapter 4 we present an application case for this dissertation in terms of a real cluster-based web application, which focus on power and performance management concerns. In Chapter 5 we present an experimental evaluation of our approach. Finally, Chapter 6 presents conclusions and outlines directions for future work.

2 Related work

In the recent years, several approaches have been proposed for dynamic adaptation of applications from different perspectives [9, 10, 15, 19, 28, 30, 31, 46, 60]. In general, similarly to our proposal, related works assume a control loop of some form to *monitor* and *adapt* a target application. In this chapter we present and discuss some of the most related ones to our approach.

2.1 CR-RIO

Many of the design concepts of our approach are based on the previous works presented on [21, 41] by members of our research team, where the CR-RIO framework was proposed and developed. Originally, the adaptation language provided by CR-RIO has been defined in terms of architectural contracts with features to specify the execution context and resource requirements for self-adaptive applications. Also, these contracts guide configuration adaptations on the application’s architecture, which are supported by a middleware infrastructure [21]. However, these contracts only provide a set of generic primitive architectural operators for adding and removing components and connections (e.g., instantiate, remove, link, etc.). Although their approach may be a useful basis for formal verification capabilities of the contracts [68], it limits the ability to meet more complex adaptation needs for applications. The problem is the large semantic gap between what is described in the contract level and what will be applied during the execution of the contracts. This way, expressing a more complex adaptation logic becomes impracticable, or most of the important adaptation decisions are left exclusively to the implementation stage.

For example, in the contracts, the different configuration options (or execution contexts) for an application are defined in terms of a state-based representation, where each state represents an application configuration. As in each configuration option (or state) we can only use architectural primitive operations, we need a vast number of states to

deal with complex adaptation needs such as specifying a decision/configuration algorithm for selecting heterogeneous components from a (possibly dynamic) set. As such, for each choice option, we would need a state representation, leading to a state explosion problem.

For the particular problem above, where the point is to simply select a particular component to be integrated or removed from the application’s architecture, some advances were made by our research group. Initially, a selection operator was described in [12] as part of a resource monitoring and discovery proposal in the context of the CR-RIO framework. Then, the work presented in [39, 40] carried out several experiments involving the use of the selection operator (based on utility functions [27]) to choose the best access point device in a wireless network for pervasive applications. In this sense, we have presented a similar proposal in the context of Grids in terms of a general decision function for selecting resources, which was proposed and integrated to the CR-RIO framework and also based on utility functions [52]. We also made some progress to investigate and generalize the two cases of selection: (a) individual components and (b) application’s architectural configurations. In [50], we proposed a general utility-based selection model. As we shall see, the proposed utility-based model is used in our current approach to decide between (conflicting) adaptation options (Chapter 3).

From another standpoint, our approach presents a contract-based adaptation language, where higher-level adaptation operators can be defined by means of *adaptation scripts* to exploit details from the intended implementation context (or style) of the applications (e.g., cluster-based client-server) [49, 51]. The design of our adaptation language is intended to support application style-specific operations which allows representing the adaptation logic at a higher granularity than primitive architectural operations, making it easier to satisfy a wide range of adaptation requirements. In addition, it helps to cope with bridging the gap between the architectural level to the implementation level. In other words, the point is to provide means for reducing the distance between the adaptation requirements and the code which actually implements these requirements.

2.2 CASA

The CASA approach [46] provides support for the development and management of adaptive applications. In CASA, the adaptation policy of every application is defined in a so-called application contract, which is stored externally to the adaptive application and based on XML. The contract is intended to describe how the application should be

reconfigured according to changes in the executing environment. CASA includes a set of adaptation mechanisms for different adaptation perspectives: (1) change in lower-level services, (2) weave and unweave of aspects in application code using AOP (aspect-oriented programming) approaches, (3) recomposition of application components, and (4) change of application attributes. We provide an extensible adaptation infrastructure, based on reflexive facilities incorporated in dynamic/scripting languages, which allows advanced users to develop new adaptation mechanisms and operators, and introduce them into our framework. Thus, we believe that our approach is able to cover these adaptation perspectives in a fairly uniform way.

2.3 Rainbow

Our proposal adopts similar elements to those described in the Rainbow framework [16, 26]. Rainbow presents an architecture-based approach for dynamic adaptation, where software architecture models are used for monitoring and effecting dynamic changes to an application. The adaptation strategies are defined at the architecture level by means of *scripts* written using a language called “Stitch”, which presents a formal declarative way of stating adaptations. The Stitch language also provides an utility-based approach to decide between different options of adaptations, while considering multiple factors or quality objectives of concern to the application [16, 18]. We adopt an analogous approach in our contracts, where a negotiation construct can be used to choose the best adaptation from a set of applicable ones (Chapter 3).

Also, our adaptation infrastructure has some similarities to Rainbow, which is divided into different layers: *system*, *architecture*, and *translation*. Run-time monitoring and adaptation of applications are carried out at the *system* layer. The adaptation decisions are taken at the *architecture* layer and are based on rules and invariants associated with the adaptation strategies. These are, in essence, equivalents to the adaptation *profiles* and *contracts* in our approach (Chapter 3).

The system layer and the architecture layer operate at different levels of abstraction, thus the *translation* layer is provided to bridge the gap between these two different abstraction levels by mapping the information exchanged between system and architecture layers. We adopt the same basic principle of the overall Rainbow approach: to separate the adaptation mechanisms, that are used to perform the dynamic adaptation work, from the adaptation logic (or policy), which represents the instructions or steps that instruct

the adaptation mechanisms what to perform.

We differ from their approach (and also from CR-RIO and CASA) in that we present a lightweight adaptation infrastructure based on the dynamic/scripting language Python (instead of the static language Java), which provides built-in reflexive mechanisms for implementing dynamic adaptation capabilities. In addition, we design our adaptation language based on these dynamic capabilities using Python scripts¹. From the point of view of adaptation script writer, Python is a well known, dynamic, easily learned, while *Stitch* (Rainbow’s adaptation language) is a new language, requiring the user/developer to learn another language to express adaptation concerns. We argue that using highly dynamic languages, we simplify the development of our adaptation infrastructure, while providing proper abstractions and flexibility for our adaptation language.

2.4 JADE

JADE is a framework for the development of autonomic management systems, which is based on the Fractal reflective component model [9]. The controlled application is described in terms of components provided with elementary management capabilities. This description, in turn, is the base of the feedback control loops that implement various self-management functions, e.g., to cater for quality-of-service requirements. Control loops provide means to link probes (sensors) to reconfiguration (actuators) mechanisms, in order to implement autonomic behaviors. In Jade, legacy applications are managed by wrapping them into components. For example, Jade has been initially used for the administration of clustered J2EE applications [9].

A critical shortcoming of the Jade framework is the lack of an adaptation knowledge representation in terms of a well-defined adaptation language, such as our contracts. Also, besides catering for quality-of-service requirements, our framework facilitates the support of power management techniques, such as those describe in [24, 59], using our contract-based adaptation language. In particular, one of the contract examples used in this dissertation (Chapter 4) is based on a decision and configuration function which uses power management optimization techniques similar to those presented in [5, 6].

¹For example, our contracts are written in a Python-like language, which allows us to compile at run-time the code of the adaptation scripts into a code object, and execute them, using Python’s built-in functions, such as `compile` and `exec` [44].

2.5 IBM autonomic systems

The IBM's Autonomic Computing initiative [35] envision autonomic computing systems to be able deal with increasing software and environment complexity, thanks to the self-managing characteristic of these systems. Such self-managing systems (divided into self-configuring, self-healing, self-optimizing and self-protecting) are managed by *autonomic managers* that monitors the element, analyzes it and its environment for potential problems, plans actions, and executes changes in a control loop fashion [32].

In recent years, some approaches from the IBM group have been proposed to turn this vision into reality, while coping with power management concerns. In [34], for instance, the authors propose a middleware to exploit power management (in terms of a control loop that allows the CPU frequency and voltage to be reduced) based on load balancing at the level of entire server clusters. In [22], the authors demonstrate a prototype for a power management solution integrated to a data center context. Their work employs server management tools, appropriate sensors and monitors, and an agent-based approach to achieve specified power and performance objectives. For example, by turning off servers under low-load conditions, the proposed approach achieved over 25% power savings over the case where the approach was not used, without incurring quality-of-service penalties in web-server clusters.

According to [36, 69], adaptation policies are typically represented as “condition-action” rules or high-level utility functions. Using the utility-based approach, an application developer or administrator specifies an utility function to determine the “desired” states (or configurations) of the application. Then, the adaptation infrastructure automatically adapts the application configuration towards higher utility. This approach is commonly used in the context of dynamic resource selection and allocation, where user's preferences are encoded by those utility functions [40, 52, 58, 57, 69].

The problem with these proposals from IBM, however, is that they are based solely on utility functions to represent the adaptation policies. Since our adaptation framework is intended to be applicable in a broad scope, including scripting level adaptations, we argue that basing only on the utility approach is not suitable to represent a wide range of adaptation requirements. Thus, we adopt a hybrid approach utilizing “condition-action” rules and utility functions for representing the adaptation expertise in terms of an adaptation language, as proposed in the work of [16, 18].

2.6 Summary

In this chapter we have summarized and discussed some of the works done by other researchers that we consider the most related to our proposal. We have shown how our approach compares to these related works and contributes in the research context of system adaptation. For example, our contract-based adaptation language has some similarities with Rainbow's adaptation language, whereas the Jade approach lacks a well-defined adaptation knowledge representation. In the next chapter, we shall describe the elements of our proposal, which mainly consists of an adaptation infrastructure with reusable elements and an adaptation language to enable one to express high-level adaptation policies.

3 The framework

Our framework adopts an external control loop paradigm [15], which works on top of an abstract architectural model. The framework provides mechanisms to (1) specify and monitor run-time properties of an executing application, (2) evaluate the model for application’s requirements violation and (3) perform adaptations to maintain the application within acceptable bounds of behavior. In our case, dynamic adaptations in the applications are carried out in accordance with an external adaptation logic specified in terms of a high-level contract-based adaptation language [51].

3.1 Software architecture model

An architectural perspective shifts focus away from source code to coarse-grained components and their interconnections. This allows designers to abstract away obscure details and focus on the application structure and interactions among components [66]. In our approach, we adopt an architecture-oriented model that allows programming the software configuration of applications [21, 41]. In this model, *components* encapsulate the application’s functional aspects and represent the main computational elements and data stores of the system: clients, servers, databases, etc. Components themselves may represent complex systems (e.g., server clusters), which are represented hierarchically as sub-architectures. *Connectors* are used in the architecture level to define relationships between components; in the operation level connectors mediate the interaction between components. *Ports* identify access points through which components and connectors provide or require services.

Furthermore, we assume that architectural elements may be annotated with various important *properties*, as described in [16]. For example, properties associated with a connector might define its protocol of interaction (e.g., HTTP). Properties associated with a component may define its core functionality, some performance attributes (e.g., average time to process a request, load, etc.), as well as cost attributes (e.g., power consumption).

Part of these properties needs to be monitored dynamically, thus our approach provides a software infrastructure to monitor the target application and map run-time application changes into architectural changes (cf. Section 3.3). The supporting infrastructure also needs to maintain the model updated on each adaptation (change) performed in the target application.

In this work, we adopt an abstract design for representing the application’s software architecture. We assume that the software architecture models — described in terms of some architectural description language, e.g., CBabel [67] or Acme [63] — are translated to an object-oriented model. From this model we are primarily interested in extracting information about the *components* (and their *connections*) and *properties* of interest of the application. This information is crucial to represent the abstract state and behavior of the application at run-time, enabling to reason about policies of adaptation. We assume that the initial configuration for the application is given to our framework as a meta-level data, which can also guide the deployment and evolution of the application. Therefore, the adaptation contracts can reference the components, types, and properties in the application’s configuration.

3.2 Contract-based adaptation language

In our approach, to express the application-specific adaptation knowledge, we can define *contracts* associated with the application’s architecture model. The contract-based adaptation language is defined by the following elements:

- *Profiles* represent conditions (Boolean expressions) associated with architectural properties and can be used to define a predicate that determines whether one or more architectural properties (e.g., server’s utilization) are valid (e.g., above some threshold). Typically, profiles are used to identify application conditions for triggering adaptations;
- A set of *adaptation scripts* captures the adaptation logic that needed to be performed to restore the normal behavior of the application; that is, moving it away from an undesirable condition. For example, a cluster-based web application may have a performance profile (e.g., server’s utilization above some threshold); upon validating it, an adaptation script associated with this profile may increase the application’s performance by adding a new server in the cluster configuration;

- Finally, a *negotiation* construct may be specified to explicitly establish a particular order (or priority) to deploy the adaptations scripts, which is defined by a transition system (state machine) [41]. Alternatively, the transitions can be driven by utility functions associated with the adaptation scripts [50]. The utility-based approach provides support to choose the best adaptation script from a set of applicable ones, while considering multiple factors or quality objectives for a particular domain of concern to the application (Section 3.2.1).

The adaptation scripts in the contracts use *abstract* adaptation operators which are mapped to application-specific operators at run-time (using the *Actuator* modules at application layer, cf. Section 3.3). The idea is to describe a skeleton so that the adaptation policy is described in abstract terms, externalizing the adaptation logic from the concrete actions to adapt a particular application. For example, the application-level operators may be as primitive as operating system calls to stop and start processes, or may be specifically built using APIs provided by the application support level (e.g., an extension module for the Apache web server using its own API [70]). Also, for each application quality attribute (e.g., load) used in the profiles, an application-specific *Sensor* module is assumed to be available or has to be developed in order to obtain (monitor) the respective run-time values from the application execution environment.

The syntax ¹ of the proposed adaptation language is defined as follows:

```

<statement> ::= (<profile_stmt> | <contract_stmt>)+
<profile_stmt> ::= profile '{' <expr> '}' <name> ';'
<policy_stmt> ::= contract '{' (<adapts>)+ '}' <name> [<adapt_period>] ';'
<adapts> ::= adaptation '{' <script> '}' <name> with <name> [<stl_time>] ';'
<adapt_period> ::= adapt_period <number>
<stl_time> ::= settling_time <number>

```

The <expr> can be any expression, like “server.load > 150”; <name> represents all alphanumeric characters plus the underscore, like “response_high”; <number> represents integer numbers, like “12”. The <script> represents the adaptation source code; preferably, written in some dynamic scripting language. Currently, it is written as Python scripts (Section 3.4).

¹The notation is the usual extended BNF, in which $\{a\}$ means 0 or more a 's, $[a]$ means an optional a , and $(a)^+$ means one or more a 's.

The execution semantics for the adaptation language is as follows. For each contract, all the profiles associated to it (specified using the **with** operator) are tested every **adapt_period** value (if not explicitly specified, default is 5 seconds)². This period value expresses a trade-off between responsiveness and overhead. As we shall see, in terms of our experimental examples, values in the order of a few seconds were found suitable. Once an adaptation option in the contract specification is applicable to be triggered (that is, its associated adaptation *profile* matches the system state), then its adaptation script code is actually interpreted and executed. In cases where more than one adaptation option is applicable to be selected and executed, our adaptation language support an utility-based negotiation approach (Section 3.2.1).

Another issue addressed in our adaptation language semantics is related to the proper time to perceive the adaptation effect to avoid oscillation between two (or more) competing adaptation options; for example, one adaptation to *increase* the capacity for a server and another to *decrease* its capacity based on server load monitoring values. Specifically, we assume that each adaptation statement can be specified with a time window to indicate how long to *wait* before we could expect to observe the stabilized conditions of the executed adaptation (by using the **settling_time** operator). If the time window value for the **settling_time** operator is well dimensioned, it would enable to achieve a beneficial lagging effect between triggering adaptations. In [17], a similar technique was adopted to address this particular aspect. Note that we can also smooth out high short-term fluctuations by increasing the adaptation periodicity (using the **adapt_period** operator). Furthermore, as we shall see, our framework approach is designed to provide a flexible and extensible support for developing and using different *Filter* modules for the profiles, like exponential moving average and predictive-based ones.

In practice, each contract has one thread of control of the following form:

```
while contract.running:
    for each a in contract.adaptations:
        if a.profile is True:
            execute adaptation code of "a"
            sleep for "settling_time" interval
        sleep for "adapt_period" interval
```

²Note that this period value should be greater or equal than the actual sampling period for the variables that need to be monitored in the application-specific layer.

```

01 contract {
02   adaptation { ... } addServer with highResponseTime;
03   adaptation { ... } increaseFreq with highResponseTime;
04 } contractExample;

```

Figure 2: A contract example with similar adaptation alternatives.

3.2.1 Utility-based adaptation negotiation

In our approach, one contract represents the adaptation knowledge for a specific domain of concern, such as power management or fault tolerance. Within one contract, we can have multiple adaptation scripts that address a particular adaptation condition (e.g., high cluster load). When the adaptation scripts address the same concern, we use utility functions to determine the most appropriate adaptation within a set of applicable ones [50]. For different domains of concern, such as power management vs. fault tolerance, our approach provides the required support via interacting contracts (Section 3.3.1).

To illustrate the utility-based adaptation negotiation, consider an example of a cluster infrastructure. In this example, when the server cluster quality is degraded (e.g., high response time), we have the option to (1) turn on servers in the cluster, and (2) to increase the operating frequency of the active processors³. These adaptations would help in some way to improve the cluster’s quality-of-service. A decision on the best adaptation will depend on a negotiation specification that explicitly describes the objectives and expected effects on choice of each adaptation alternative. In Figure 2, we describe the aforementioned adaptations in terms of a contract example.

To make a choice from a range of adaptation alternatives, several factors that influence the choice are modeled in terms of quality dimensions over the adaptation alternatives. In principle, a quality dimension is an attribute which is orthogonal to the application’s functionality. Examples of quality dimensions include cost, performance, security, and scalability. We assume that domain experts can provide application-specific prioritization for these quality dimensions. We denote A as the set of the adaptation alternatives and D as the set of quality dimensions. In terms of our contract example, we define $A = \{addServer, increaseFreq\}$ and $D = \{cost, performance\}$.

Specifically, using an utility-based approach, the preferences over the quality dimensions are represented as a real number w_d for each $d \in D$, such as $\sum_{d \in D} w_d = 1$. In the

³Naturally, if the response time is low, the opposite adaptations could be defined: turn off servers, and decrease operating frequency of the active servers.

form of an utility function $U(a)$, where $a \in A$, we denote

$$U(a) = \sum_{\forall d \in D} w_d \times u_d(a)$$

For each quality dimension $d \in D$, we define an utility function $u_d : a \rightarrow [0, 1]$ to have an explicit representation of the expected costs and effects of each adaptation alternative $a \in A$. An utility function for a quality dimension can be defined by discrete values or by means of analytical expressions such as linear, exponential, or logarithmic functions.

Suppose we have an utility function to represent the cost (e.g., power) for our adaptation alternatives that yields $u_{cost}(addServer) = 0.50$, $u_{cost}(increaseFreq) = 0.70$. Also, we have an utility function for performance (e.g., requests per second) that yields $u_{perf}(addServer) = 0.90$, $u_{perf}(increaseFreq) = 0.80$. Considering the preferences $w_{cost} = 0.6$ and $w_{perf} = 0.4$, we can score the adaptation alternatives as follows:

$$U(addServer) = 0.6 \times 0.5 + 0.4 \times 0.9 = 0.66$$

$$U(increaseFreq) = 0.6 \times 0.7 + 0.4 \times 0.8 = 0.74$$

Therefore, the *increaseFreq* adaptation ranks higher by using the utility function score.

3.3 Adaptation support infrastructure

The supporting infrastructure of our approach is depicted in Figure 3. This infrastructure is composed by a standard set of entities:

- CONTRACT MANAGER (CM) interprets the *contracts* and extracts from them the information regarding the *adaptations scripts*, respective *profiles*, and *negotiation* specification. Periodically, profiles associated with adaptation scripts are evaluated and an adaptation may be triggered if the actual system state matches the respective profile. In the case of two or more conflicting adaptation options, the decision on which adaptation script to use is guided by the negotiation specification;
- CONTRACTOR manages and mediates the run-time monitoring process of the properties specified in the profiles. The Evaluator interacts with *Sensors* to obtain and evaluate the measured values of the properties. Violations and validations of the

profiles are notified to the CM. To deal with transient or stochastic properties, we rely on *Filter* modules attached to *Contractor*. For example, a particular filter module can either adopt an exponential moving-average (EMA), which is a well-known technique to smooth out measurement readings, or use a predictive filtering technique to identify *trends* in measurements. The latter intends to enable anticipatory adaptations [55] (Section 5.6);

- CONFIGURATOR is the element responsible for mapping adaptation scripts into actions that adapt the application architecture. The Configurator interacts with *Actuators* that represent the individual mechanisms necessary to implement application-specific adaptation actions, e.g., allocation of a new server in a cluster. *Translators* entities (attached to Configurator) are used to help with the mapping of information across the abstraction gap from the *reusable layer* to the *application layer*. For example, to map an architectural-level element identifier into a hostname or an IP address.

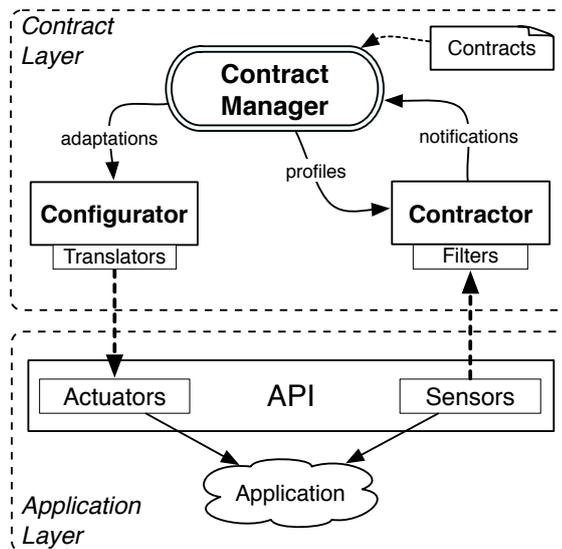


Figure 3: Adaptation framework.

The adaptation infrastructure provides thus a set of reusable mechanisms to interpret (*Contract Manager*), monitor (*Contractor*) and impose (*Configurator*) the adaptation contracts. In our approach, each contract can represent a specific adaptation logic. However, they can share common monitoring functions and adaptation operators used in the contracts. Once the application-level sensors and actuators have been developed, one can reuse those entities in other adaptation contracts. For example, for the contracts that use the same quality attributes (e.g., cluster load) in the profiles, all the monitoring

mechanism can be reused. In the same way, the contract operators to adapt the target application can be reused in other contracts that share the same concern (e.g., power management), which means reusing common operators to turn on and off machines, and to adjust (i.e., increase or decrease) the CPU frequency of the servers. For the contracts that use different quality attributes (e.g., response time) or different adaptation contexts (e.g., multimedia application), only the new application level sensors or actuators would have to be developed.

3.3.1 Supporting multiple contracts

Our approach provides support for multiple domains (or concerns) of adaptations by composing multiple contracts. To achieve this, we rely on a concurrency model⁴ which is the key to the management of the contracts. In this model, each *contract* is represented by an *autonomous entity* and the contracts (or threads) can access the shared application/architecture model. To avoid adaptation interferences, we adopt a global locking mechanism between the contracts; that is, each contract needs to obtain the shared lock in order to begin its execution (which consists in a control loop). We need this to ensure consistency across the architecture/application model.

By using the global locking mechanism, we can also resolve potential conflicts between the contracts. The basic idea is analogous to the *First-Come, First-Serve* approach, which is adopted for conflict resolution among adaptation strategies, motivated by its agility (fast responsiveness), in a similar distributed self-adapting framework [30].

For instance, suppose we have two adaptation policies: one contract for power management called *PM*, and another contract *FT* for fault tolerance. In the case of the adaptation contract *FT* is ready to be executed when the contract *PM* is being executed, the locking mechanism prevents the contract *FT* to execute. After finishing the execution of the contract *PM*, the blocked contract *FT* is released and retried immediately. It is worth mentioning that the profiles associated with the contract *FT* are constantly reevaluated (while it is blocked) by the *Contractor* module. We consider that the execution time of the contract *PM* (including its settling time, cf. Section 3.2) is sufficient to enable the *Contractor* module to monitor and update the system properties. In this way, the conditions for adaptation (expressed in the profiles) remain consistent with the current system state.

⁴Actually, we adopt Python's threading module which is based on Java's thread model, with some minor differences [44].

In the application example presented in Chapter 4, we define two adaptation contracts for different concerns: (1) power and performance management and (2) fault tolerance in the server cluster. By using our approach, we are able to compose these two contracts in order to support adaptation for multiple concerns via interacting contracts. To evaluate this ability of composition, we show an experiment using the two coordinating contracts in Chapter 5.

3.4 Implementation details

We have implemented a prototype of our adaptation framework. The contract layer of reusable elements were developed in the object-oriented dynamic language Python 2.4 [44]. The architecture models and the contracts are translated by our adaptation infrastructure and stored as meta-level data (object model) associated to the application. In the current implementation, we have adopted the XML-RPC protocol as the standard transport infrastructure between the architecture/contract reusable layer and the application-specific layer.

The benefits of using scripting/dynamic languages for coordinating applications while maintaining their core in a compiled/static languages have long been discussed elsewhere [64]. Usually, scripting languages are used for connecting software components together. The Python language, for instance, offers support for integration with other languages and tools. A wide variety of tools provided by the standard library, combined with the ability to use a lower-level language such as C and C++, makes Python a powerful *glue language*, which helps to bridge the gap between our reusable adaptation infrastructure layer and application-specific layer (as illustrated in Figure 3).

As a dynamic language, Python provides means to evaluate expressions at run-time, like `eval('cluster.load > T_HIGH')`, which turns to be very useful to simplify the implementation of the *Contractor* module of our adaptation infrastructure. Also, the Python language explicitly provides support for dynamic reconfiguration of components, while some other languages (e.g., Java) need to be extended (with new language constructs) for supporting these dynamic capabilities. As an example, the work presented in [62] have used Java technologies (e.g, Javassist [20], a class library to manipulate byte-codes in Java) to support dynamic adaptations in the context of the CR-RIO framework.

The contracts are written in a Python-like language, which allows us to compile the code of the adaptation scripts into a code object, and execute them, using Python's

built-in functions, such as `compile` and `exec` [44]. Moreover, Python’s design philosophy emphasizes developer productivity and code readability — its core syntax and semantics are minimalist, while the standard library is large and comprehensive. We argue that highly dynamic languages (as Python) provide proper abstractions and flexibility for representing the adaptation logic to be performed on the applications.

3.5 Summary

In this chapter we have described our approach in terms of its contract-based adaptation language and reusable infrastructure. The adaptation language is provided to enable one to express application-specific adaptation policies, while the adaptation infrastructure provides a set of reusable mechanisms to interpret, monitor and effect those adaptation policies. In the next chapter, we present an application case for our approach in the context of a real cluster-based web application, which focus on power and performance management, and fault tolerance concerns.

4 Application case

In this chapter, we describe how the elements of our framework are used to support dynamic adaptation capabilities for a cluster-based application, which needs to maintain two correlated objectives. First, we aim to guarantee some quality-of-service requirements for the cluster (e.g., by controlling average cluster load or request response time). Second, in order to reduce costs, the set of currently active servers and their respective processor's speeds should be configured to minimize the power consumption. We also present an adaptation policy to achieve fault tolerance requirements in the cluster, intended to be used in conjunction with other contracts representing different adaptation concerns (e.g., power management).

4.1 Architecture description

The architecture (shown in Figure 4) consists of a cluster of replicated web servers. The cluster presents a single view to the clients through a special component termed *ServerCluster* (also called *front-end* or *load balancer*), which distributes incoming requests among the actual *Servers* that process the requests (also known as *back-ends* or *workers*).

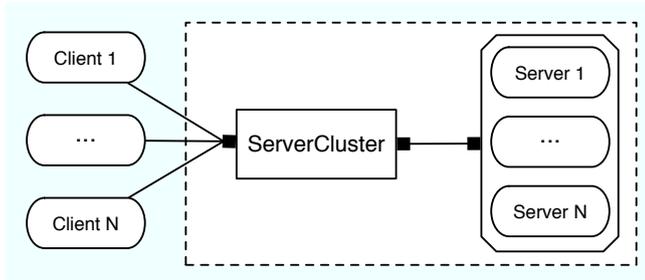


Figure 4: Architecture of the web application.

The architecture description is translated to an abstract representation of the current cluster configuration. Specifically, a cluster configuration is represented by a set of tuples, where each tuple is defined by $(i, j) \in conf$, where $i \in N$ represents a server component

reference, N is the number of total servers in the cluster, and $j \in \{-1, 0\} \cup F_i$ represents the state of server i , where F_i is the set of discrete frequencies (steps) available in the server processors (e.g., $\{1, 2, 3, 4\}$). If j is equal to zero, then server i is inactive, otherwise server i is active and its processor is operating at frequency j . We also consider the special case of server failure, when j is set to -1 . We define an auxiliary function $maxFreq : i \rightarrow |F_i|$ to return the maximum frequency of server i , and function $state : i \rightarrow j$ to return the current state of server i . One example of a cluster configuration, where $N = 3$, is $conf = \{(1, 0), (2, 3), (3, 2)\}$, which means that server 1 is turned off, server 2 is operating at frequency 3, and server 3 at frequency 2. In the architecture model, we represent the current cluster configuration as a property (termed *currentConf*) of *ServerCluster* component.

4.2 Adaptation contracts

In order to guarantee the application’s quality requirements, we specify a set of *adaptation scripts* to be performed in response to changes in the execution environment of the application (e.g., workload variation). A simple and effective way to identify an undesirable state in which adaptations should be carried out is to define bounds for specific application quality properties, such as cluster utilization above $X\%$ or requests response time above Y seconds. The quality properties are defined using *profiles* in the contract specification. Note that our framework is designed to enable one to describe in profiles any quality attribute for which bounds can be defined. Even for quality attributes that do not have straightforward numeric measurements, it is often possible to derive a numeric or discrete state representation (e.g., probability or percentage) for which bounds can be defined, such as percentage of request deadlines met.

We measure the cluster quality in terms of the cluster utilization, which refers to the ratio of the *actual* number of requests per second (req/s) received by the cluster to the *maximum* number of requests that the current cluster configuration is able to process per second. In order to keep the cluster utilization within acceptable levels, we define two profiles (shown in Figure 5) which uses two thresholds `U_LOW` and `U_HIGH`, corresponding to the cluster under-utilization and over-utilization, respectively.

These profiles are then monitored at run-time by the supporting infrastructure (Section 3.3). Note that for each quality attribute (e.g., load or responseTime) used by the profiles, an application-specific *Sensor* module is assumed to be available or has to be

```

profile {
  webcluster.load / webcluster.maxLoad() < U_LOW
} lowUtil;

profile {
  webcluster.load / webcluster.maxLoad() > U_HIGH
} highUtil;

```

Figure 5: Profiles expressing the quality bounds for the cluster utilization.

developed in order to obtain (monitor) the respective run-time values from the application execution environment.

During the execution of the clustered web application, so as to respond to web requests properly when the server cluster quality is degraded, we have the option to (1) turn on servers in the cluster and (2) to increase the operating frequency of processors. As well, to save energy, we can (3) turn off servers in the cluster and (4) reduce the operating frequency of the processors, in the case of the server cluster is providing a much higher-quality service than that required by the applications.

A decision on the best adaptation will depend on a logic (or policy) that describes the objectives and expected effects on choice of each adaptation alternative. In real server clusters, as in our case, processors can be heterogeneous, adding to increase the number of configuration possibilities. For example, when the cluster utilization is high, we have to decide the best choice between (a) turning on new servers or (b) increase the operating frequency of the currently active servers, or (c) an optimized combination of adaptations (e.g., turning off a particular server and turning on another with a higher capacity, but more energy efficient). Next, we present two contract examples for power and performance management in server clusters.

4.2.1 Simple power management contract

The first example of adaptation contract (described in Figure 6) adopts a simple idea which attempts to increase (or decrease) the active servers frequencies up to a maximum (or minimum) and only then turn servers on (or off). It uses two adaptation scripts termed *increaseCapacity* (lines 02-13) and *decreaseCapacity* (lines 15-25), which are associated with profiles *highUtil* (line 13) and *lowUtil* (line 25) to identify the precise conditions for adaptations — in this example, the cluster utilization bounds.

According to this contract, the profiles are evaluated at every *adaptation_period* (see

```

01 contract {
02   adaptation {
03     s = webcluster.nextServerToIncFreq()
04     if s != None:
05       curfreq = s.state
06       webcluster.adjustServerFreq(curfreq + 1)
07     else:
08       s = webcluster.nextServerToAdd()
09       if s != None:
10         webcluster.turnServerOn(s)
11       else:
12         log("no more servers to turn on")
13   } increaseCapacity with highUtil settling_time 3000/*ms*/;
14   adaptation {
15     s = webcluster.nextServerToDecFreq()
16     if s != None:
17       curfreq = s.state
18       webcluster.adjustServerFreq(curfreq - 1)
19     else:
20       s = webcluster.nextServerToRem()
21       if s != None:
22         webcluster.turnServerOff(s)
23       else:
24         log("no more servers to turn off")
25   } decreaseCapacity with lowUtil settling_time 3000/*ms*/;
26 } simple adaptation_period 5000 /*ms*/;

```

Figure 6: A simple dynamic power management policy.

line 26) by the adaptation infrastructure to check if there are any adaptations to be performed. The choice for the adaptation period value represents a trade-off between responsiveness and overhead — also in terms of intermittent disruptive reconfigurations (Section 3.2). In our contract examples, values in the order of a few seconds were found suitable.

For each adaptation option, we specify a time window to indicate how long to wait before we could expect to observe the stabilized conditions of the executed adaptation (see *settling_time* operators in Figure 6 — lines 13 and 25). If the time window value for the *settling_time* operator is well dimensioned, it would enable to achieve the desirable delay (lag) effect between triggering adaptations, avoiding oscillation between two (or more) competing adaptation options. Note that the thresholds used in the profiles should be selected based on an appropriate range to help prevent oscillatory behavior.

This contract uses the following operations to query the state of the application ar-

chitecture model:

- *nextServerToIncFreq()* / *nextServerToDecFreq()*: These operations find and return a server reference in the current cluster configuration which has a frequency feasible to be increased (or decreased) discretely. In the case of not finding any server (i.e., all active servers are at maximum or minimum speed), *None* is returned. Below, we show a possible implementation for these operations;

```
def nextServerToIncFreq():
    for s in power_servers:
        if 0 < s.state < s.maxFreq:
            return s
    else:
        return None
```

```
def nextServerToDecFreq():
    for s in reversed(power_servers):
        if s.state > 1:
            return s
    else:
        return None
```

- *nextServerToAdd()* / *nextServerToRem()*: These operations are used to find and return a server reference in the current cluster configuration available to be turned on (or off). In the case of not finding any server, *None* is returned. Below, we show an implementation for these operations.

```
def nextServerToAdd():
    for s in power_servers:
        if s.state == 0:
            return s
    else:
        return None
```

```
def nextServerToRem():
    for s in reversed(power_servers):
        if s.state == 1:
            return s
    else:
        return None
```

The above adaptation operators are based on a previously ordered list of servers (*power_servers*). In particular, the servers are ordered by power efficiency, which is defined by the ratio of power consumption vs. performance, where power consumption is measured in *watt* and performance in *requests per second* (req/s). That is, the servers are increasingly ordered by those which consume less energy per request (i.e., *joule / req*). The operation *nextServerToIncFreq*, for example, iterates over the *power_servers* list — ordered in ascendent order — to find a server reference which is turned on and is not operating at maximum frequency.

The next adaptation operators effect changes to the application architectural model:

- *turnServerOn(server) / turnServerOff(server)*: These operations are used to turn machines on and off in the cluster. In practice, the load balancing mechanism must be aware of the new state of the servers in the cluster (i.e., booting or shutdown) so that it does not redirect requests to inoperable servers;
- *adjustServerFreq(server, freq)*: This operation dynamically adjusts the frequencies of servers in the cluster. As we can dynamically change the computational power of servers, the load balancer must adopt a policy to effectively balance the load among servers; for example, a dynamic weighted round robin (DWRR) scheme.

To implement those operations, our adaptation infrastructure relies on *actuators* entities (cf. Section 3.3), which are application-specific modules, already available in the application execution environment (in some form of an API) or specifically built for controlling the target application. In our case, we have developed the actuators as an extension module using the Apache web server module API [70], as we shall see in Section 4.4.

4.2.2 Decision-based power management contract

The second contract (shown in Figure 7) presents a more elaborated logic based on a decision function to dynamically set up the best cluster configuration. That is, the adaptation contract decides which servers must be active and their operating frequencies, while tackling the current workload demand and minimizing the cluster power consumption. Differently from the simple contract previously presented, we define an adaptation script named *adjustCapacity* (lines 02-13), which joins the two adaptations *increase/decreaseCapacity* concepts into one. The profiles *lowUtil* and *highUtil* — used to identify when to trigger the adaptation based on cluster utilization — are merged (by operator "or") and associated with the respective adaptation script (see line 13).

In the specification of the *adjustCapacity* adaptation (lines 02-12 in Figure 7), we use the current cluster load to calculate the demand, while regulating it by a factor taking into account the maximum cluster utilization threshold; specifically, we correct the demand value by normalizing it to the `U_HIGH` preset limit to achieve the desired cluster utilization bound (see line 04). Next, we execute the *bestConfig* function (line 04) to select the best cluster configuration, according to the load demand and a configuration model. Then, a loop (lines 05-11) is used to impose the chosen configuration over the application's architecture.

```

01 contract {
02   adaptation {
03     demand = webcluster.load / U_HIGH
04     changeConf = webcluster.bestConfig(demand)
05     for (server, freq) in changeConf:
06       if freq == 0:
07         webcluster.turnServerOff(server)
08       else:
09         if server.state == 0:
10           webcluster.turnServerOn(server)
11           webcluster.adjustServerFreq(server, freq)
12   } adjustCapacity with lowUtil or highUtil settling_time 6000/*ms*/;
13 } decision adapt_period 5000/*ms*/;

```

Figure 7: Decision-based policy for dynamic power management.

The *bestConfig* adaptation operator is defined abstractly and specifically many optimization algorithms or techniques could be used to solve the cluster configuration problem. For this particular contract, we have formulated the cluster configuration problem as a mixed integer program (MIP), which is expressed as follows:

$$\text{Minimize } \sum_{i \in N} \sum_{j \in F_i} \alpha_{ij} \times p_busy_{ij} + \beta_{ij} \times p_idle_{ij} \quad (4.1)$$

$$\text{Subject to } \alpha_{ij} + \beta_{ij} = X_{ij} \quad \forall i \in N, \forall j \in F_i \quad (4.2)$$

$$\sum_{j \in F_i} X_{ij} \leq 1 \quad \forall i \in N \quad (4.3)$$

$$\sum_{i \in N} \sum_{j \in F_i} \alpha_{ij} \times perf_{ij} \geq demand \quad (4.4)$$

$$\alpha_{ij}, \beta_{ij} \in [0, 1], X_{ij} \in \{0, 1\} \quad \forall i \in N, \forall j \in F_i \quad (4.5)$$

The values of *p_idle* and *p_busy* are related to the active and the baseline (or idle) power consumption cost of each member of the set of servers and their respective frequencies, whose information was extracted using a power measurement approach similar to that presented in [6]. Specifically, we have built a *Sensor* entity in our framework, deployed on a dedicated machine in the cluster, for power measurement using the LabVIEW software environment [47], which relies on a USB based data acquisition (DAQ) device (the USB-6009 from National Instruments [48]). The DAQ device was configured to acquire the power measures — taken from the AC power — for the servers in the cluster. The values of *perf* are based on the measured performance of the servers, for each

frequency, in terms of the maximum number of *requests per second* (req/s) that they can handle at 100% CPU utilization, for each server i and respective frequency j . To generate the benchmark, we used the *httperf* [45] tool. This power and performance information is crucial for solving the problem of selecting the best cluster configuration.

The goal of the objective function given by formula (4.1) is to find a cluster configuration that minimizes the overall power consumption, while dealing with the incoming workload (given by the *demand* parameter in the constraint equation (4.4)), where α_{ij} represents the utilization factor and β_{ij} represents the idle factor for a given server i and respective frequency j . The solution is given by the decision variable X_{ij} , as shown in constraint (4.2), where i is a server reference and j is the server's status (i.e., its operating frequency or inactive status, when $j = 0$). The constraint (4.3) is defined so that it can be chosen only one frequency j on a given server i . Using this mathematical formulation, the *bestConfig* adaptation operator can thus be implemented by a wide variety of standard optimization algorithms.

In this application example, we have implemented the *bestConfig* operator using the Python module PyGLPK [25]. This module encapsulates the functionality of the GNU Linear Programming Kit (GLPK) [42]. The cluster configuration model was written in the GNU MathProg language, which is a subset of the AMPL language supported by GLPK. After solving the MIP optimization problem, the *bestConfig* operator returns a cluster configuration solution given by notation $(server, freq) \in solutionConf$, which represents a configuration of servers and their respective status (i.e., its operating frequency or inactive). Actually, the cluster configuration to be imposed is a difference between the two sets: current cluster configuration and the cluster configuration solution. For example, suppose the current configuration is $currentConf = \{(1, 0), (2, 2), (3, 1)\}$ and the solution configuration is $solutionConf = \{(1, 0), (2, 1), (3, 4)\}$. Thus, we need to impose a configuration change given by $solutionConf - currentConf = changeConf = \{(2, 1), (3, 4)\}$. That is, we need to decrease the frequency of server 2 to 1, and increase the frequency of server 3 to step 4.

During initial experiments using this decision-based contract, we observed a peculiar transition between two cluster configurations: If the current cluster configuration has only one active server and a new configuration performs a swap operation between servers, which means to deactivate the current server and activate on a new one, a problematic behavior may occur. That is, if the new configuration shutdown the current server before the new server is ready to respond the requests, as the server booting time is not instan-

taneous, the cluster will be in an unavailable state. To solve this problem, we simply need to modify the new configuration representation so that the operation to shutdown servers is always performed at the end. This works because the operation to activate servers blocks until receives an event that server is active, or an error occurs.

In fact, the cluster configuration problem has exponential complexity with regard to the number of servers. Thus, depending on the number of servers, more advanced optimization approaches (e.g., heuristics) or state-of-art commercial optimization packages (e.g., CPLEX) would be necessary, in order to make decision at run-time. Another way would be to solve the optimization problem offline and store the solution into a table to be looked-up at run-time, as proposed in [59]. However, this approach is not flexible enough for our further work, which is to be able to dynamically change (expand or shrink) the set of servers in the cluster, in the context of a data center, where concurrently clusters can share all the server's processors. For this case, we could specify a higher level adaptation contract that would manage the overall processor allocation, based on the cluster configuration problem formulation presented here.

4.2.3 Fault-tolerance contract

In a cluster-based system, when a server fails, the service remains available due to the inherent replication. However, the quality-of-service might become noticeably degraded. A solution is to replace the faulty server by a new one. To achieve this, we specify a repair policy as a new contract for our cluster-based application (see Figure 8). First, we identify which server is failed (line 03). Second, we use an operator to designate a new server to be added in the cluster (line 04). Finally, we use an operator to actually replace the failed server with the new one, if the new server is correctly assigned to the cluster.

```

01 contract {
02   adaptation {
03     srv = webcluster.getFailedServer()
04     newsrv = webcluster.allocNewServer()
05     if newsrv: webcluster.replaceServer(srv, newsrv)
06     else: webcluster.log("could not allocate server")
07   } replaceServer with serverFailure settling_time 1000/*ms*/;
08 } repairPolicy adapt_period 1000/*ms*/;

```

Figure 8: A contract for fault-tolerance.

This contract uses a new profile (shown in Figure 9) to identify when a server has

failed. Also, the *failure* property (when greater than zero) gives the identification i of the failed server (i.e., $0 < i < N$, where N is number of servers in the cluster).

```

profile {
  webcluster.failure > 0
} serverFailure;

```

Figure 9: Profile for identifying a server failure.

In this application example, we intend to use this contract in composition with other contracts representing different adaptation concerns, e.g., the power management contracts presented earlier in this chapter. Following the separation of concerns paradigm, this would keep the power management and the fault management contracts independent, which improves modularity and reusability. Notice that the adaptation period of the fault tolerance contract (Figure 8) is shorter than, e.g., the power management contract (Section 4.2.2). In practice, this means that we are interested into giving more “priority” to the fault tolerance contract in order to detect failure much faster than to optimize energy consumption.

4.3 Reusability and flexibility

In our approach, each contract represents a specific adaptation logic. However, given an application domain they can share common monitoring functions and contract operators. Once the application-level sensors and actuators have been developed, one can reuse those entities in other adaptation contracts. For example, for the contracts that use the same quality attributes (e.g., cluster load) in the profiles, all the monitoring mechanism can be reused. In the application-layer, we reused the *sensor* employed to measure the cluster *load* (hence, the utilization) property associated with *webcluster* component type.

In the same way, the contract operators to adapt the cluster configuration can be reused in other contracts that share the same concern (such as power management), which means reusing the *turnServerOn*, *turnServerOff*, and *adjustServerFreq* operations. For contracts that use different quality attributes (e.g., cluster response time) or different adaptation concerns (e.g., multimedia stream application), only the new application level sensors or actuators would have to be developed.

4.3.1 Contracts using different quality metrics

In order to explore the flexibility of our approach, we define another way to measure the cluster quality, in terms of the average request response time, which refers to the time interval measured between the instant of that the request arrives at the cluster and the time that the associated response leaves the cluster. In order to keep this new quality attribute within acceptable levels, we need to define two profiles (shown in Figure 10), using the `RT_LOW` and `RT_HIGH` as the quality thresholds. We denote the high threshold as the request deadline.

```

profile {
  webcluster.responseTime < RT_LOW
} lowRT;

profile {
  webcluster.responseTime > RT_HIGH
} highRT;

```

Figure 10: Profiles expressing the cluster response time limits.

Next, we define two new contracts derived from the previous ones, which was: (1) the simple contract, (2) the decision-based contract. We call these new adaptation contracts as: *simpleRT* and *decisionRT*. These contracts shall use the same profiles (i.e., `highRT` and `lowRT`), so it means that the basic elements for monitoring can be reused for both contracts. Also, in the application-specific layer, we can reuse the *sensor* used to effectively measure the cluster *responseTime*, which is a property associated with *webcluster* component type.

The *simpleRT* contract is shown in Figure 11. The only change we made in this contract was to rename the profiles, using now the *highRT* and *lowRT* profiles (see lines 13 and 25).

On the other side, the *decisionRT* contract (shown in Figure 12), which relies on current cluster load to calculate the demand for the *bestConfig* operator (line 07), has now also to be aware of the current average request response time. It means that some specific code has to be written in the adaptation script to cope with the new quality requirement. Specifically, in the case of the cluster response time rises above the predefined quality bound (i.e., when profile *highRT* is valid), we still must guarantee that the response time restriction is met (see line 04). Thus, we determine the *tardiness* (line 05) by the ratio of current response time to the maximum acceptable response time (i.e. deadline), which means “how far” we are from the request deadline. Then, we regulate the demand by

```

01 contract {
02   adaptation {
03     (...)
04   } increaseCapacity with highRT settling_time 3000/*ms*/;
05   adaptation {
06     (...)
07   } decreaseCapacity with lowRT settling_time 3000/*ms*/;
08 } simpleRT adaptation_period 5000 /*ms*/;

```

Figure 11: Simple contract using the request response time as the quality metric.

multiplying the current demand by the tardiness value in order to achieve the desired response time bounds (see line 06).

```

01 contract {
02   adaptation {
03     demand = webcluster.load
04     if highRT.valid:
05       tardiness = webcluster.responseTime / RT_HIGH
06       demand = demand * tardiness
07     (The code below is the same from the previous contract , but now using new profiles)
08     changeConf = webcluster.bestConfig(demand)
09     for (server, freq) in changeConf:
10       if freq == 0:
11         webcluster.turnServerOff(server)
12       else:
13         if server.state == 0:
14           webcluster.turnServerOn(server)
15         webcluster.adjustServerFreq(server, freq)
16   } adjustCapacity with lowRT or highRT settling_time 4000/*ms*/;
17 } decisionRT adapt_period 5000/*ms*/;

```

Figure 12: Decision-based contract using the request response time as quality requirement.

In order to improve even more the modularity between the original decision-based contract and the new one (based on response time metric), we can define a new adaptation function (with the steps to adapt the cluster) to be used in both contracts, as shown in Figure 13.

```

def adapt_cluster(demand):
    changeConf = webcluster.bestConfig(demand)
    for (server, freq) in changeConf:
        if freq == 0:
            webcluster.turnServerOff(server)
        else:
            if server.state == 0:
                webcluster.turnServerOn(server)
            webcluster.adjustServerFreq(server, freq)

```

Figure 13: Example of a reusable adaptation function.

4.4 The application-specific layer

For the cluster-based web application case study, we have implemented an application-specific layer using Apache 2.2.8 web servers and Linux Gentoo 2.6 operating system. The servers members of the cluster (or *back-end* servers) are standard Apache web servers that perform the same service (i.e., all servers can process all requests). The application-specific entities for monitoring (*Sensors*) and for effecting adaptation (*Actuators*) were developed in the Apache web server *front-end* based on its built-in proxy load balancer module (named `mod_proxy_balancer`) [70]. The `mod_proxy_balancer` module has a default request counting scheduler algorithm to distribute the HTTP requests among the cluster servers and uses *lb factors* (or weights) to assign the servers' work quota. This is a normalized value representing their contribution to the amount of work to be accomplished. If one server has *lb factor* 2 while second has *lb factor* 1, than the first server will receive two times more requests than second one. If the *lb factor* of a server is zero, it is considered to be disabled. Once a server is disabled, it can be put into a low-power state (e.g., suspended to RAM). Note that this sequence of actions is reversed if we plan to turn on the servers.

More explicitly, we implemented the application-level *sensors* and *actuators* by means of a new Apache module called *mod_frontend*, which uses the Apache proxy load balancer module functionalities. For example, to enable (or disable) the servers in the cluster as well as to dynamically assign weights (termed *lb factors*) to the servers. In this module, we expose a generic interface (through the XML-RPC protocol) to monitor the cluster quality properties (e.g., load) and to manage the web clustered application (e.g., to turn servers on/off and to adjust servers frequencies). For example, to enable (or disable) a server in the cluster (setting *lb factor* = 0) as well as to dynamically assign weights (*lb factor* > 0) to a server. To monitor the *failure* property used in the fault-tolerant

contract (Section 4.2.3), we relied on the servers' status (termed *lbstatus*) available on a shared data structure maintained by the Apache's load balancer module.

As we can dynamically change the computational power of servers, we calculate the *lbfactor* of a server i as $lbfactor_i = freq_i \times K_i / 1000$, where $freq$ is given in *MHz* and K is a constant related to the server hardware features. In our cluster, the servers are heterogeneous with respect to different numbers of available and maximum frequencies, respectively. However, for a given frequency, they have the same performance, because of their common hardware architecture characteristics. Thus, we simply define K as the number of CPU cores in the server, that is, $K = 1$ for single core processor, $K = 2$ for dual core, and so on. In practice, this *lbfactor*s setting strategy, proportional to the servers' performance, was found suitable to achieve a good load balancing for the incoming requests among all active servers in our cluster. A very similar approach was adopted by [6], which used the same server machines as that used in this work.

4.5 Summary

The application case described in this chapter is of crucial importance to understand how the elements of our approach works. In essence, we have shown how the adaptation contracts can be used to express dynamic adaptations capabilities for a cluster-based web application, which includes power management and fault tolerance techniques. Based on these adaptation contracts, we shall present in the next chapter an experimental evaluation for our framework-based approach.

5 Experimental evaluation

In this chapter, we present an experimental evaluation of our framework. We carried out several experiments on a dedicated cluster testbed consisting of 6 machines (one front-end web server and five back-end web servers) communicating over 1-gigabit-per-second network switch (as illustrated in Figure 14). One extra client machine was used to simulate request loads (web browser stateless sessions) from multiple clients. We have simplified our web cluster model in that we assume that there is no state information to be maintained for multiple requests within each section. Our adaptation software infrastructure was deployed in the front-end machine. It included the code for monitoring run-time properties and for performing adaptations on the cluster-based application.

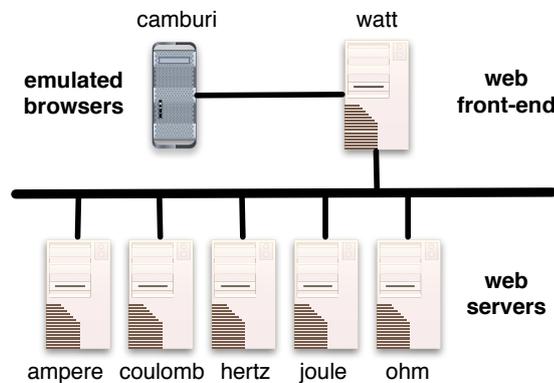


Figure 14: Testbed network topology.

5.1 Server cluster setup

Table 1 shows a detailed specification of the servers used to build our cluster¹. The processors of the servers are heterogeneous in terms of maximum performance and number of frequencies available, and all servers have 2GB of main memory (RAM). The power

¹The front-end machine (*watt*) has the same hardware characteristics as that of the *joule* server, except its maximum frequency which is 2Ghz.

consumption of a server, for each available discrete frequency, varies linearly with its CPU utilization, and we consider different idle and busy power values for each frequency. To collect the power measures for the machines in the cluster, we used a power *Sensor* built in our framework using the LabVIEW software environment [47], which relies on a USB based data acquisition (DAQ) device (the USB-6009 from National Instruments [48]). We also measured the performance of the servers, for each frequency, in terms of the maximum number of requests per second (req/s) that they can handle at 100% CPU utilization, which is named $perf_{ij}$, for each server i and respective operation frequency j . To generate the benchmark workload, we used the *httperf* [45] tool.

In this work, we use the term *load* to refer to the actual number of requests that a server is processing per second (req/s) and *utilization* to refer to the fraction of a server's total capacity that is being used. That is, the utilization of a server i at frequency j is given by the actual *load* divided by $perf_{ij}$. We also define *cluster utilization* as the ratio of *load* divided by the sum of $perf_{ij}$ such as i is an *active* server and j is its operating frequency step. For example, suppose our cluster configuration of active servers is $conf = \{(1, 2), (5, 3)\}$, which means server 1 (ampere) at frequency 1800 MHz and server 5 (ohm) at frequency 2000 MHz. According to Table 1, if the incoming load is 290 req/s, then our cluster has utilization of $290/(168.4 + 184.4) = 82\%$. Note that this definition of cluster utilization assumes the web requests are CPU-bound, which is reasonable for most servers because much of the required data are already in main memory [59].

Server 1: ampere CPU: AMD Athlon(tm) 64 X2 3800+				Server 4: joule CPU: AMD Athlon(tm) 64 3500+			
Freq. (MHz)	P _{busy} (W)	P _{idle} (W)	Perf. (Req/s)	Freq. (MHz)	P _{busy} (W)	P _{idle} (W)	Perf. (Req/s)
1000	81.5	66.3	94.7	1000	74.7	66.6	47.1
1800	101.8	70.5	168.4	1800	95.7	73.8	84.0
2000	109.8	72.7	187.6	2000	103.1	76.9	93.5
				2200	110.6	80.0	102.0
Server 2: coulomb CPU: AMD Athlon(tm) 64 3800+				Server 5: ohm CPU: AMD Athlon(tm) 64 X2 5000+			
Freq. (MHz)	P _{busy} (W)	P _{idle} (W)	Perf. (Req/s)	Freq. (MHz)	P _{busy} (W)	P _{idle} (W)	Perf. (Req/s)
1000	75.2	67.4	47.5	1000	82.5	65.8	92.9
1800	89.0	70.9	84.6	1800	99.2	68.5	165.9
2000	94.5	72.4	94.0	2000	107.3	70.6	184.4
2200	100.9	73.8	102.8	2200	116.6	72.3	201.0
2400	107.7	75.2	111.5	2400	127.2	74.3	218.1
				2600	140.1	76.9	235.3
Server 3: hertz CPU: AMD Athlon(tm) 64 3800+							
Freq. (MHz)	P _{busy} (W)	P _{idle} (W)	Perf. (Req/s)				
1000	71.6	63.9	47.0				
1800	85.5	67.2	83.9				
2000	90.7	68.7	93.0				
2200	96.5	69.9	102.3				
2400	103.2	71.6	110.5				

Table 1: Power and performance specification of the servers in our cluster.

5.2 Implementation issues

To build our server cluster, we have used servers with Dynamic Voltage and Frequency Scaling (DVFS). This is a technique that consists of varying the frequency and voltage of the microprocessor in run-time according to processing needs. Also, we have used machines that can suspend their execution to RAM (STR, *Suspend-to-RAM*) to save power, commonly referred to as *standby* or *sleep*. In this state, main memory (RAM) is still powered, although it is almost the only component that is [2]. As observed in [6], the power consumption when suspended to RAM is about 5.5 watts. This is worth spending because the boot time increases from $\sim 7s$, when the server is suspended, to $\sim 30s$, when the server is halted. To resume from the suspended state, the servers support the *Wake-on-LAN* mechanism, which allows a machine to be turned on (or woken up) remotely when receiving a special network message.

Initial tests using our framework showed some practical issues when executing operations to adapt the cluster-based application. The problem was that in the case of servers running close to 100% CPU usage (although not meant to happen), the operations triggered to repair the servers (e.g., increase CPU frequency) were delayed for too long. Thus, some quality metric of the web cluster (e.g., utilization or response time) becomes noticeably degraded. The solution found for this problem was to run the process which executes local adaptations with a higher priority than the main web server process. Since our adaptation infrastructure is running under Linux operating system, we were able to change the scheduling policy of the adaptation effector process to `SCHED_RR` (Round Robin scheduling), which is intended for time-critical applications. By using this technique, we could greatly reduce the response time of effecting adaptation operations in our cluster infrastructure.

To help avoid false triggering adaptations, we have implemented in our framework a *filter* module (called *ExpFilter*) based on a single exponential moving-average [1], smoothing out high short-term fluctuations in measurements readings. Specifically, the *ExpFilter* computes the next value, S_t , by summing the product of the smoothing constant α ($0 < \alpha < 1$) with the new value (X_t), and the product of $(1 - \alpha)$ times the previous average, as follows: $S_t = \alpha * X_t + (1 - \alpha) * S_{t-1}$. Values of α close to 1.0 have less smoothing effect and give greater weight to recent changes in the data, while values of α close to 0.0 have a greater smoothing effect and are less responsive to recent changes. That is, the choice of α allows the engineer to control the speed at which the measures are smoothed, without shifting the application reaction time too far from its intended

target. Simple exponential smoothing is easily applied, and produces a smoothed value as soon as two observations are available. Sometimes the engineer’s judgment is used to choose an appropriate smoothing factor, or alternatively some techniques may be used to optimize the value of α , such as using the Marquardt procedure to find the value of α that minimizes the mean of the squared errors (MSE) [1]. In the *ExpFilter* module, we have used $\alpha = 0.5$ as the default smoothing factor. Based on preliminary experiments, this value was found suitable.

5.3 Workload generation

In our experimental evaluation, all the workloads were generated using the *httperf* tool [45], by starting successive sessions with a fixed rate, where each HTTP request is a PHP script with an average execution time of 10.6 *ms*. This execution time was measured while all servers were turned on at full speed. Specifically, the following command was used: `httperf --server=watt --port=81 --uri=/home.php --wsess=100,5100,0.14 --rate .15`, which causes *httperf* to generate a total of 100 sessions at a rate of .15 session per second (1 session per approximately 6.6 seconds), each session consisting of 5100 calls that are spaced out by 0.14 seconds. This produces a linear ramp up and then a linear ramp down of requests, approximately 25 minutes in duration.

5.4 Effectiveness and flexibility

In order to demonstrate the effectiveness of our adaptation framework, we performed some experiments using the web application case study from Chapter 4. The first experiment (shown in Figure 15) conducted dynamic adaptations on the clustered web application in accordance with the decision-based contract description from Section 4.2.2. As previously presented, this contract employs an adaptation logic based on a decision function to dynamically choose the best cluster configuration. That is, it decides which processors should be active (or inactive) and their respective operating frequencies, while handling the current workload and minimizing the overall power consumption of the cluster.

In Figure 15, the upper plot shows the workload generated using the *httperf* tool [45], following the shape of a linear ramp up and down of requests. The middle plot demonstrates that our approach was effective in controlling the cluster quality metric (for

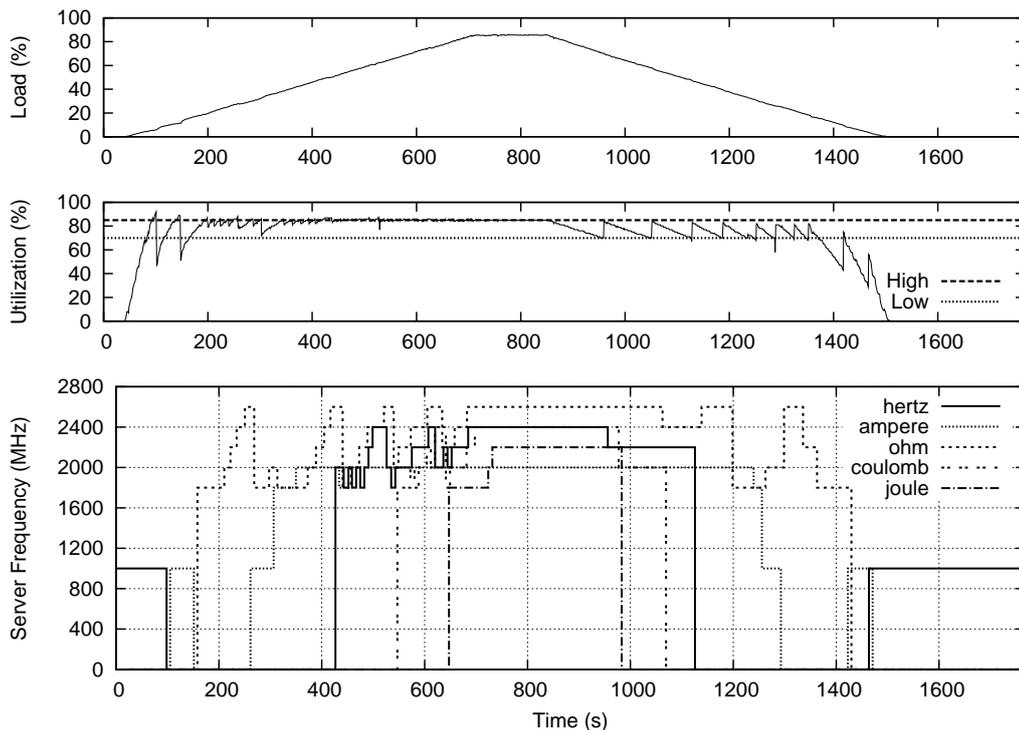


Figure 15: Execution of dynamic adaptations in accordance with the decision-based contract.

this contract example, the cluster utilization), which has varied, for most of the time, in between 70% and 85%. These values are, respectively, the quality thresholds U_LOW and U_HIGH used for the profiles of the contract specification. In the bottom plot, we can actually observe the configuration changes (frequency switching) for all servers in the cluster. If the operating frequency for a server is zero, it is considered to be turned off.

In another experiment shown in Figure 16, we measured the effectiveness of our approach in terms of the percentage of energy consumption reduction in the cluster as compared to not using our approach — that is, when all servers are turned on at full speed to handle peak load and dynamic adaptations are not conducted. This experiment shows that while using our approach, the energy consumption in the cluster is substantially reduced. In the execution 1 (the control), a total energy consumed was $690,650 J \approx 191.85 Wh$, while in the execution 2 (using our approach) was $434,508 J \approx 120.69 Wh$. It means an energy consumption reduction of $\approx 37\%$.

5.4.1 A comparison between adaptation policies

Here we describe a comparison between the adaptation contracts. For this comparison, we use the energy consumption and disruption as quality metrics. We define *disruption*

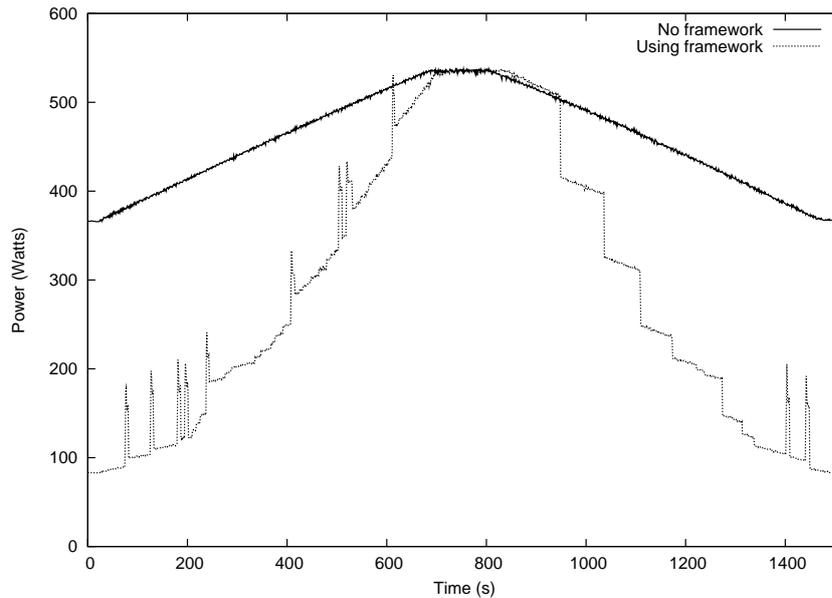


Figure 16: Energy saving in the cluster using the decision-based contract.

as the number of turning on (and off) adaptations, which may involve a switching cost that should be carefully considered in the cluster. Table 2 shows the measured values for these two metrics. As we can see, the “none” policy is the one that consumes most energy compared to “simple” (Section 4.2.1) or “decision-based” (Section 4.2.2), but it causes no disruption at all. The “decision-based” contract presents a reduction of $\approx 12\%$ over the “simple” contract, but the disruption is 2.5 times greater.

policy/contract	energy consumption	disruption
none*	690,650J	0
simple	493,582J	8
decision-based	434,508J	20

* all servers are turned on at full speed to handle peak load.

Table 2: A comparison between different adaptation contracts.

In the Section 5.6, we discuss some reasons for these disruptions and outline a viable solution which would be using prediction-based techniques. Although our work is not meant to address this particular issue, the presented framework provides useful basis to help reduce some of these disruptive scenarios.

5.4.2 Using different cluster quality metric

In another experiment (shown in Figure 17), we demonstrate the flexibility of our approach by enabling a different cluster quality attribute to be controlled. Specifically,

we used the decision-based adaptation contract that controls the cluster response time requirement (termed *decisionRT*, cf. Section 4.3.1). This example demonstrates how easily a developer can apply a different quality metric to be controlled using our approach.

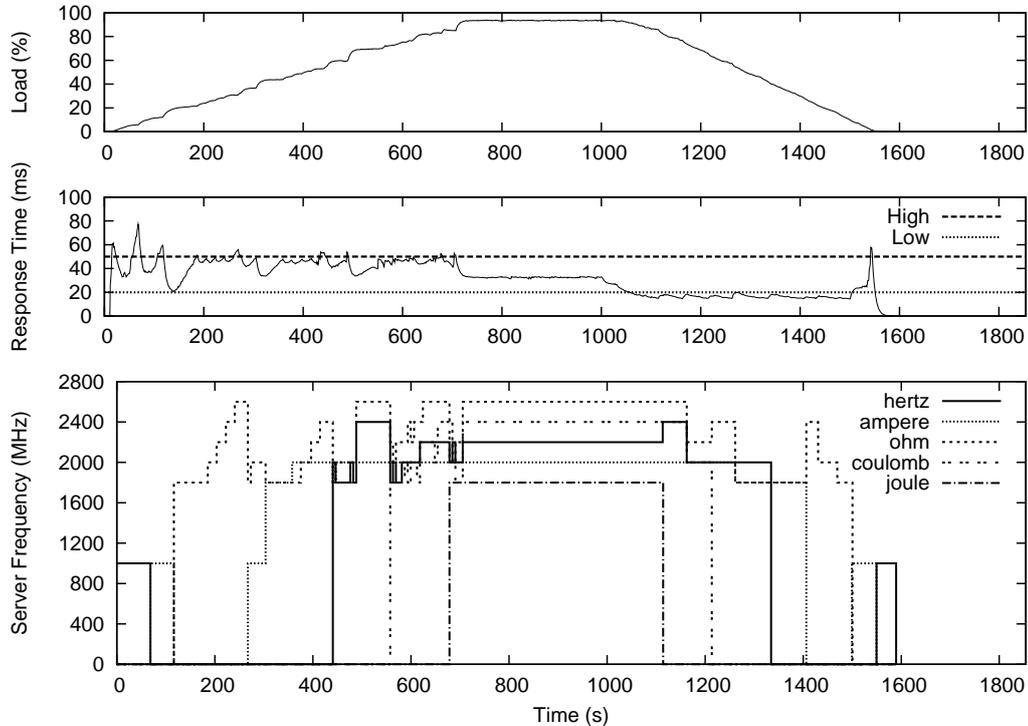


Figure 17: Execution of a new contract using the request response time as quality requirement.

In Figure 17, the upper plot shows the workload generated using the *httperf* tool. Because of the request response time measure was shown to be much more unstable than the cluster load measure (that was used in the previous experiments), we had to adjust the smoothing factor of the *ExpFilter* to $\alpha = 0.2$. Doing so, we could more efficiently control the response time quality requirement. As shown in the middle plot, the cluster response time has varied, for most of the time, in between $20ms$ and $50ms$. These values are, respectively, the quality thresholds RT_LOW and RT_HIGH used for the profiles of the adaptation contract (as described in Section 4.3.1). In the bottom plot, we can actually observe the cluster configuration changes (frequency switching). When the operating frequency for a server is zero, it is considered to be turned off.

5.5 Composing multiple contracts

To evaluate the ability of our approach in composing multiple contracts, we carried out an experiment using two contracts that address two different domain of concern: one

for power management (Section 4.2.2) and another for fault tolerance (Section 4.2.3). In this experiment, the cluster was composed by four servers (instead of five): *ampere*, *coulomb*, *hertz*, *joule*, and the *ohm* server was configured to be the backup server (Table 1). We simulated a failure scenario by forcing the Apache process of the *ampere* server to stop during the experiment execution. Note that this experiment involved stateless web servers; that is, their internal state did not need to be preserved between failures.

In Figure 18, we can observe the power management and the failure tolerance contract running concurrently. The upper plot shows the workload accepted by our server cluster, where the failure was detected at time $\approx 433s$ and the system configuration was restored (at time $\approx 457s$) to normal operation after replacing the *ampere* (faulty server) by a new server one (*ohm* server) — see the bottom plot. Thus, the cluster repair phase took $\approx 24s$.

Notice that the actual load (request rate) on the cluster depends on the cluster’s capacity. If it has capacity enough, the actual rate is the one imposed by the *httperf* tool. However, if the cluster capacity is below that needed to cater for the imposed rate, the cluster’s processed response rate drops. This is what happens during the cluster repair phase (upper plot, Figure 18).

In the middle plot, we observe that the cluster utilization shows an oscillatory behavior (some peaks of $\approx 100\%$ usage) during the repair phase. After the repair activity, a few seconds were necessary to stabilize the system. In the sequel, after the system settles down to its new configuration, the cluster utilization is effectively controlled by the power management contract, until the end of the experiment.

5.6 Opportunity for anticipatory adaptation

From 0s to 200s in the bottom plot of Figure 15, it may be observed that the *hertz* server had to be turned off for the *ampere* server to be turned on. In the sequel, the *ampere* server had to be turned off to allow the *ohm* server to be turned on. This is an example of disruptive adaptations that are not desired because they mean waste of processing time (and certainly energy), mainly in the medium/long term. Specifically, these situations occur because the adaptation decisions are determined by an optimization function (*bestConfig*) in the contract specification, which uses only instant snapshots of the inputs (e.g., current load) to make a decision. The problem with this strategy is that several locally optimal reactive adaptations may often be less than optimal over time [55]. Here the opportunity arises to improve the global efficiency of the adaptation

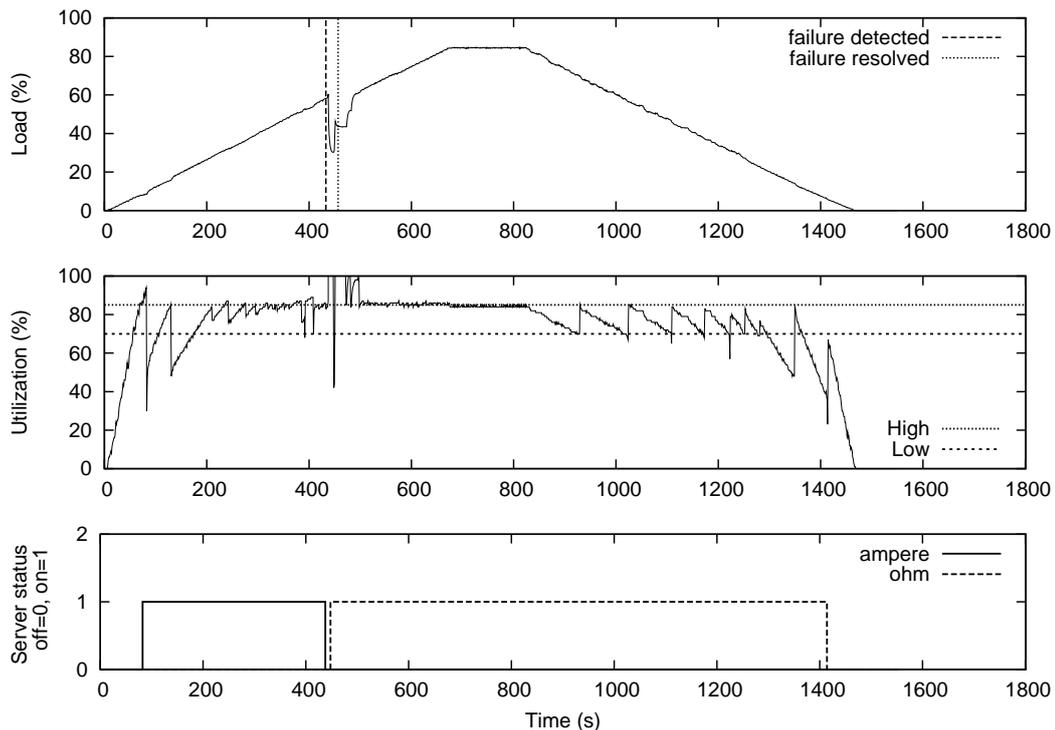


Figure 18: Concurrent execution of the contracts: power management and failure tolerance.

logic employed by the application; that is, using more suitable filter modules in our framework. One could implement new filter modules with predictive capabilities in order to cope with *trends* in measurements readings that indicate anticipatory conditions for triggering adaptations, which may prevent undesirable oscillatory behavior. For instance, the Holt forecasting procedure, a variant of exponential smoothing, could be used [1, 61]. This is a simple and widely used method that copes with trend and can be suitable for producing short-term forecasts for demand time-series data, such as the incoming cluster workload, although it does not deal with seasonality or periodicity, like Holt-Winters [14]. The design of these predictive filters is outside of the scope of this dissertation. Our framework, however, provides a customization point to cater for predictive approaches in order to improve adaptation decisions [3, 23, 54].

5.7 Adaptation timing analysis

In order to evaluate a possible overhead impact incurred using our framework, we identify two crucial parts of the whole adaptation process that would be interesting to measure, which are: (1) to evaluate the profiles (*monitor*) and (2) to execute the adaptation scripts (*adapt*). To evaluate the profiles, we rely on sensors to obtain the run-time

properties of an executing application; then, we apply a filter procedure, and next we update the object model with the evaluation result of the profiles. To effect the adaptations, we rely on the actuators which are associated with the adaptation operators used in the contract description. We assume that the *decision* phase, between the monitor and adapt phases, has a controllable and relatively short processing time which can be disregarded in this analysis.

Based on the previous experiments, our measurements show that the *monitoring* phase takes on average $1.27ms$ with standard deviation (stdev) of $0.67ms$. For the worst case execution, $6.06ms$, most of the time was spent in the execution of the sensor modules, $4.08ms$, which represents about 67% of the overall monitoring time. The *adaptation* phase takes on average $876.57ms$, with stdev of $2,585.48ms$. This high standard deviation is due to the high differences among the times associated to the adaptation operators. For instance, to actually turn on a server, it takes on average $7,979.80ms$, with stdev of $1,550.26ms$; to turn a server off takes $1,010.85ms$, with stdev $3.03ms$; and to adjust server frequency takes $7.09ms$, with stdev $1.63ms$. Table 3 summarizes the execution time of each adaptation step.

The worst case measured for the overall adaptation phase was $13,045.78ms$, while the actual adaptation operations was: one *turn_on*, one *turn_off*, and then two *adjust_freq*; these operations took respectively $12,012ms + 1,005ms + 7ms + 7ms = 13,013ms$ in the application-level. Thus, we observe that most of the time was spent during the actual execution of the adaptation operators, and the net overhead of using the framework was only $32.78ms$. In the context of the typical adaptation period for server clusters (in the order of seconds or few minutes), our approach uses a suitable adaptation time window, while incurring small processing time overhead.

Adaptation steps	exec. (#)	min (ms)	max (ms)	mean (ms)	stdev (ms)
(1) Evaluate profiles	1,523	0.45	6.06	1.27	0.67
(2) Effect adaptations	209	1.04	13,045.78	876.57	2,585.48

Sub-step from “Evaluate profiles” (application-level):

(1.1) get remote values	1,524	0.28	4.08	0.76	0.44
-------------------------	-------	------	------	-------------	------

Sub-steps from “Effect adaptations” (application-level):

(2.1) turn server on	20	6,106.0	12,013.0	7,979.80	1,550.26
(2.2) turn server off	20	1,005.0	1,018.0	1,010.85	3.03
(2.3) set server freq.	154	5.0	13.0	7.09	1.63

Table 3: Execution times of the adaptation steps.

5.8 Summary

To evaluate our approach, we have carried out several experiments on a dedicated cluster testbed. These experimental results demonstrate that our approach is useful and effective in providing the required support for describing and deploying typical power management and fault tolerance contracts. Specifically, for the case of power optimization, our approach allowed to achieve energy savings of 37%, while meeting the application's quality-of-service requirements. Also, we have shown that the processing time overhead of the framework is not significant.

6 Conclusion

In this dissertation, we have presented a framework-based approach which was used to support dynamic adaptation capabilities in a web server cluster environment. By experimental evaluation, we have demonstrated that our framework was useful and effective in providing the required support to express typical adaptation policies for power management and fault tolerance, through the use of high-level adaptation language, as well as to efficiently deploy these adaptive policies using our adaptation infrastructure.

6.1 Contributions

The main contributions of this dissertation consist in providing a framework with a reusable adaptation infrastructure and an adaptation language to support dynamic adaptation in the context of web server clusters. By using external adaptation infrastructure, our approach enables one to modify and reason about different application's adaptation policies and quality-of-service requirements. Our approach also provides support for multi-objective adaptations by composing multiple adaptation policies. The adaptation infrastructure can be reused across different adaptation requirements, which helps to reduce the cost of engineering such adaptive applications.

6.2 Future work

Although we have focused on dynamic adaptations in a distributed server cluster scenario, the implementation we have presented here is essentially centralized, with monitoring and adaptation being performed within a single adaptation infrastructure instance. However, depending on the application needs, the components of the framework could be deployed in a distributed setting (with replicated module instances) to deal with concerns regarding scalability and single-point failure. By having distributed module instances, the framework design should be aware of concurrent access to the same shared applica-

tion distributed model [16]. This is an issue that will need to be investigated in a future project.

Taking advantage of our framework capabilities, we intend to use different prediction techniques [54, 56], implemented through filter modules, which are themselves easily changeable using the framework. This would enable our adaptation infrastructure to make anticipatory decisions, helping to improve the overall behavior of the controlled application. In the presented cluster application, unnecessary (power-consuming) switching of processors, which may occur during a short low activity period, could be avoided, leading to more energy savings and less application disruption.

We are also investigating the use of contracts to cater for additional requirements of real server architectures, such as support of multi-layer servers [29], where each layer can be managed by a specific contract, and more elaborate contracts for server fault management, required to meet availability requirements. In a data center scenario [13], several server clusters can be active at the same time, each one associated to a different service (and managed by its own individual contract). In this context, a higher level contract can manage the overall processor allocation over the set of server clusters. This would allow to expand or to shrink the set of processors of each cluster in order to attend peak or low demand periods of activity. Assuming that resource demands vary along time, this could lead to further performance and energy consumption optimizations.

Further, in the data center case, we see an interesting future investigation in exploiting dynamic adaptation techniques in the context of server virtualization, which has been successful at reducing the number of servers in server clusters and helping reduce their power consumption [37]. For example, our framework would provide the basic software infrastructure to monitor utilization (or any other quality metric) across the server clusters and effect configuration operations on the virtualized server environment, e.g., to power off unneeded physical servers without impacting applications and users.

The coordination, specification and implementation of such autonomous adaptation contracts (working concurrently), intended for attaining an overall adaptation goal, poses a great challenge for future research. For example, for achieving consistent behavior, conflicts for the use of the shared resources and on the execution of conflicting adaptation policies have to be overcome. Based on initial investigations in this work on supporting multiple contracts, we believe that the elements of our adaptation framework provides the basis to help solve these issues.

In the experiments presented in this dissertation, we have used a simple workload

generation scheme using the *httperf* tool in order to show the effectiveness of our approach. For future experiments, we intend to use the web-log traces available at the Internet Traffic Archive [38], e.g. WorldCup98, which provides a more realistic web server workload characterization to be investigated for our experiments.

References

- [1] Introduction to time series analysis, Section 6.4, NIST/SEMATECH e-handbook of statistical methods. <http://www.itl.nist.gov/div898/handbook/>, 2008.
- [2] ADVANCED CONFIGURATION AND POWER INTERFACE. ACPI Specification. <http://www.acpi.info/>, 2008.
- [3] BARYSHNIKOV, Y., COFFMAN, E. G., PIERRE, G., RUBENSTEIN, D., SQUILLANTE, M., AND YIMWADSANA, T. Predictability of web-server traffic congestion. In *Proceedings of the Tenth IEEE International Workshop on Web Content Caching and Distribution* (Sept. 2005), pp. 97–103.
- [4] BERTINI, L., LEITE, J., AND MOSSÉ, D. Statistical QoS guarantee and energy-efficiency in web server clusters. In *19th Euromicro Conference on Real-Time Systems* (2007), pp. 83–92.
- [5] BERTINI, L., LEITE, J., AND MOSSÉ, D. Dynamic configuration of web server clusters with QoS control. In *WIP Session of the 20th Euromicro Conference on Real-Time Systems* (2008).
- [6] BERTINI, L., LEITE, J., AND MOSSÉ, D. Optimal dynamic configuration in web server clusters. Tech. Rep. RT-1/08, Instituto de Computação — Universidade Federal Fluminense, 2008.
- [7] BIANCHINI, R., AND RAJAMONY, R. Power and energy management for server systems. *Computer* 37, 11 (2004), 68–74.
- [8] BOHRER, P., ELNOZAHY, E. N., KELLER, T., KISTLER, M., LEFURGY, C., MCDOWELL, C., AND RAJAMONY, R. The case for power management in web servers. 261–289.
- [9] BOUCHENAK, S., PALMA, N. D., HAGIMONT, D., AND TATON, C. Autonomic management of clustered applications. In *IEEE International Conference on Cluster Computing* (Barcelona, Spain, 2006), IEEE Computer Society.
- [10] CAPRA, L., ZACHARIADIS, S., AND MASCOLO, C. Q-cad: Qos and context aware discovery framework for adaptive mobile systems. *International Conference on Pervasive Services (ICPS '05)* (2005), 453–456.
- [11] CARDELLINI, V., CASALICCHIO, E., COLAJANNI, M., AND YU, P. S. The state of the art in locally distributed web-server systems. *ACM Comput. Surv.* 34, 2 (2002), 263–311.

- [12] CARDOSO, L. Integração de serviços de monitoração e descoberta de recursos a um suporte para arquiteturas adaptáveis de software. Master's thesis, Instituto de Computação — Universidade Federal Fluminense, 2006.
- [13] CHANDRA, A., GONG, W., AND SHENOY, P. Dynamic resource allocation for shared data centers using online measurements. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2003), ACM, pp. 300–301.
- [14] CHATFIELD, C. The holt-winters forecasting procedure. *Applied Statistics* 27, 3 (1978), 264–279.
- [15] CHENG, B. H., GIESE, H., INVERARDI, P., MAGEE, J., AND DE LEMOS, R. 08031 – software engineering for self-adaptive systems: A research road map. In *Software Engineering for Self-Adaptive Systems* (Dagstuhl, Germany, 2008), B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds., no. 08031 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.
- [16] CHENG, S.-W. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 2008. Technical Report CMU-ISR-08-113.
- [17] CHENG, S.-W., AND GARLAN, D. Handling uncertainty in autonomic systems. In *International Workshop on Living with Uncertainties* (Atlanta, GA, USA, November 2007).
- [18] CHENG, S.-W., GARLAN, D., AND SCHMERL, B. Architecture-based self-adaptation in the presence of multiple objectives. In *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems* (New York, NY, USA, 2006), ACM Press, pp. 2–8.
- [19] CHENG, S.-W., GARLAN, D., SCHMERL, B. R., SOUSA, J. P., SPITZNAGEL, B., STEENKISTE, P., AND HU, N. Software architecture-based adaptation for pervasive systems. In *ARCS '02: Proceedings of the International Conference on Architecture of Computing Systems* (London, UK, 2002), Springer-Verlag, pp. 67–82.
- [20] CHIBA, S. Javassist — a reflection-based programming wizard for java. In Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java, 1998.
- [21] CORRADI, A. Um framework de suporte a requisitos não-funcionais para serviços de nível alto. Master's thesis, Instituto de Computação — Universidade Federal Fluminense, 2005.
- [22] DAS, R., KEPHART, J. O., LEFURGY, C., TESAURO, G., LEVINE, D. W., AND CHAN, H. Autonomic multi-agent management of power and performance in data centers. In *The Seventh International Conference of Autonomic Agents and Multiagent Systems* (May 2008).
- [23] DINDA, P. A., AND O'HALLARON, D. R. Host load prediction using linear models. *Cluster Computing* 3, 4 (2000), 265–280.

- [24] ELNOZAHY, E. N., KISTLER, M., AND RAJAMONY, R. Energy-efficient server clusters. In *Power-Aware Computer Systems* (2003), vol. 2325 of *Lecture Notes in Computer Science*, pp. 179–197.
- [25] FINLEY, T. *PyGLPK*. <http://www.cs.cornell.edu/~tomf/pyglpk/>, 2008.
- [26] GARLAN, D., CHENG, S.-W., HUANG, A.-C., SCHMERL, B., AND STEENKISTE, P. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer* 37, 10 (October 2004).
- [27] GOMES, L. F. A. M., GOMES, C. F. S., AND DE ALMEIDA, A. T. *Tomada de Decisão Gerencial — Enfoque Multicritério (2a edição)*. Editora Atlas S.A., São Paulo, Brasil, 2006.
- [28] HILLMAN, J., AND WARREN, I. An open framework for dynamic reconfiguration. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 594–603.
- [29] HORVATH, T., ABDELZAHER, T., SKADRON, K., AND LIU, X. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Transactions on Computers* 56, 4 (2007), 444–458.
- [30] HUANG, A.-C., AND STEENKISTE, P. Building self-adapting services using service-specific knowledge. In *HPDC '05: Proceedings of the High Performance Distributed Computing* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 34–43.
- [31] HUEBSCHER, M. C., AND MCCANN, J. A. An adaptive middleware framework for context-aware applications. *Personal Ubiquitous Computing* 10, 1 (2005), 12–20.
- [32] IBM. An architectural blueprint for autonomic computing. <http://www.ibm.com/developerworks/autonomic/library/ac-summary/ac-blue.html>, 2005.
- [33] JONATHAN G. KOOMEY. Estimating total power consumption by servers in the U.S. and the world. <http://enterprise.amd.com/Downloads/svrpwrusecompletefinal.pdf>, February 2007.
- [34] KEPHART, J. O., CHAN, H., DAS, R., LEVINE, D. W., TESAURO, G., RAWSON, F., AND LEFURGY, C. Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing* (Washington, DC, USA, 2007), IEEE Computer Society, p. 24.
- [35] KEPHART, J. O., AND CHESS, D. M. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.
- [36] KEPHART, J. O., AND DAS, R. Achieving self-management via utility functions. *IEEE Internet Computing* 11, 1 (2007), 40–48.
- [37] KUSIC, D., KEPHART, J. O., HANSON, J. E., KANDASAMY, N., AND JIANG, G. Power and performance management of virtualized computing environments via lookahead control. *IEEE International Conference on Autonomic Computing* (2008).

- [38] LAWRENCE BERKELEY NATIONAL LABORATORY. The internet traffic archive. <http://ita.ee.lbl.gov/>, April 2008.
- [39] LEAL, D. Suportando a adaptação de aplicações pervasivas pelo uso de funções utilidade. Master's thesis, Instituto de Computação — Universidade Federal Fluminense, 2007.
- [40] LEAL, D., AND LOQUES, O. Seleccionando o melhor ponto de acesso com base nas preferencias do cliente e na disponibilidade dos recursos da rede sem-fio. In *Simpósio Brasileiro de Redes de Computadores (SBRC 2008)* (May 2008).
- [41] LOQUES, O., SZTAJNBERG, A., CERQUEIRA, R. C., AND ANSALONI, S. A contract-based approach to describe and deploy non-functional adaptations in software architectures. *Journal of the Brazilian Computer Society* 10, 1 (July 2004), 5–18.
- [42] MAKHORIN, A. *GLPK (GNU Linear Programming Kit) version 4.36*. <http://www.gnu.org/software/glpk/>, 2009.
- [43] MARK BLACKBURN. Five ways to reduce data center server power consumption. The Green Grid. http://www.thegreengrid.org/gg_content/White_Paper_7_-_Five_Ways_to_Save_Power.pdf (February 2008), 2008.
- [44] MARTELLI, A. *Python in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.
- [45] MOSBERGER, D., AND JIN, T. httpperf – a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.* 26, 3 (1998), 31–37.
- [46] MUKHIJA, A. *CASA — A Framework for Dynamic Adaptive Applications*. PhD thesis, University of Zurich, Switzerland, 2007.
- [47] NATIONAL INSTRUMENTS. Labview platform and development environment. <http://www.ni.com/labview/>, July 2008.
- [48] NATIONAL INSTRUMENTS CORPORATION. NI USB-6008/6009 user guide and specification. <http://www.ni.com/pdf/manuals/371303e.pdf>, July 2008.
- [49] PETRUCCI, V. A framework for supporting dynamic adaptation of power-aware web server clusters. Master's thesis, Institute of Computing, Fluminense Federal University, 2008.
- [50] PETRUCCI, V., AND LOQUES, O. Suporte a adaptação dinâmica de aplicações usando funções de utilidade. In *1st Workshop on Pervasive and Ubiquitous Computing, WPUC 2007* (October 2007), SBAC-PAD 2007.
- [51] PETRUCCI, V., LOQUES, O., AND MOSSÉ, D. A framework for dynamic adaptation of power-aware server clusters. In *SAC '09: Proceedings of the 24th ACM Symposium on Applied Computing* (2009), ACM.
- [52] PETRUCCI, V., LOQUES, O., AND SZTAJNBERG, A. Seleção de recursos em grades computacionais usando funções de utilidade. In *4th Workshop on Computational Grids and Applications (WCGA)* (July 2007).

- [53] PINHEIRO, E., BIANCHINI, R., CARRERA, E. V., AND HEATH, T. Dynamic cluster reconfiguration for power and performance. In *Compilers and Operating Systems for Low Power*, L. Benini, M. Kandemir, and J. Ramanujam, Eds. Kluwer Academic Publishers, 2002.
- [54] POLADIAN, V. *Tailoring Configuration to User's Tasks under Uncertainty*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, April 2008. Technical Report CMU-CS-08-121.
- [55] POLADIAN, V., GARLAN, D., SHAW, M., SCHMERL, B., SOUSA, J. P., AND SATYANARAYANAN, M. Leveraging resource prediction for anticipatory dynamic configuration. In *First IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO-2007* (July 2007), pp. 214–223.
- [56] POLADIAN, V., SHAW, M., AND GARLAN, D. Modeling uncertainty of predictive inputs in anticipatory dynamic configuration. In *Proceedings of the International Workshop on Living with Uncertainties (IWL'07), co-located with the 22nd International Conference on Automated Software Engineering (ASE'07)*, (Atlanta, GA, USA, 5 November 2007). <http://godzilla.cs.toronto.edu/IWL/program.html>.
- [57] POLADIAN, V., SOUSA, J. P., GARLAN, D., SCHMERL, B., AND SHAW, M. Task-based adaptation for ubiquitous computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Special Issue on Engineering Autonomic Systems* 36, 3 (May 2006).
- [58] POLADIAN, V., SOUSA, J. P., GARLAN, D., AND SHAW, M. Dynamic configuration of resource-aware services. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 604–613.
- [59] RUSU, C., FERREIRA, A., SCORDINO, C., WATSON, A., MELHEM, R., AND MOSSÉ, D. Energy-efficient real-time heterogeneous server clusters. In *IEEE Real Time Technology and Applications Symposium* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 418–428.
- [60] SALLEM, M. A. S., AND DA SILVA E SILVA, F. J. The adapta framework for building self-adaptive distributed applications. In *ICAS '07: Proceedings of the Third International Conference on Autonomic and Autonomous Systems* (Washington, DC, USA, 2007), IEEE Computer Society, p. 46.
- [61] SANTANA, C., BERTINI, L., LEITE, J., AND MOSSÉ, D. Applying forecasting to interval based DVS. In *10th Brazilian Workshop on Real-Time and Embedded Systems (WTR)* (2008).
- [62] SANTOS, A. L. G. Um suporte para adaptação dinâmica de arquiteturas. Master's thesis, Instituto de Computação — Universidade Federal Fluminense, 2006.
- [63] SCHMERL, B., AND GARLAN, D. AcmeStudio: Supporting style-centered architecture development. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 704–705.

- [64] SCHNEIDER, J.-G., AND NIERSTRASZ, O. Components, scripts and glue. In *Software Architectures – Advances and Applications*, L. Barroca, J. Hall, and P. Hall, Eds. Springer-Verlag, 1999, pp. 13–25.
- [65] SHARMA, V., THOMAS, A., ABDELZAHER, T., SKADRON, K., AND LU, Z. Power-aware QoS management in web servers. In *24th IEEE Real-Time Systems Symposium (2003)*, pp. 63–72.
- [66] SHAW, M., AND GARLAN, D. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [67] SZTAJNBERG, A. *Flexibilidade e Separação de Interesses para a Concepção e Evolução de Aplicações Distribuídas*. PhD thesis, COPPE/PEE/UFRJ, Maio 2002.
- [68] SZTAJNBERG, A., AND LOQUES, O. Describing and deploying self-adaptive applications. In *1st Latin American Autonomic Computing Symposium (July 2006)*, pp. 14–20.
- [69] TESAURO, G., AND KEPHART, J. O. Utility functions in autonomic systems. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 70–77.
- [70] THE APACHE SOFTWARE FOUNDATION. Apache HTTP server version 2.2. <http://httpd.apache.org/docs/2.2/>, 2008.