Introduction of Self Optimization Features in a Selfbenchmarking Architecture

El Hachemi Bendahmane*‡ and Bruno Dillenseger*
*France Telecom RD, Orange Labs
Grenoble, France
Email: {Elhachemi.bendahmane,bruno.dillenseger}@orange-ftgroup.com

Patrice Moreaux‡
‡LISTIC, University of Savoie
Annecy, France
Email: patrice.moreaux@univ-savoie.fr

***Abstract*—Benchmarking client-server systems involve complex, distributed technical infrastructures, whose management deserves an autonomic approach. It also relies on observation, analysis and feedback steps that closely matches the autonomic control loop principle. While previous work have shown how to introduce autonomic load testing features through self-regulated load injection, this paper sketches the path to full self-benchmarking, introducing self-optimization features to get meaningful results. Our contribution is twofold: completion of a component-based architecture, combining several autonomic control loops, to fully support self-benchmarking, and an original constraints programming-based optimization algorithm. The relevance of this work in progress is partially evaluated through first experimental results.**

***Keywords*-benchmarking; autonomic computing; self-optimization; constraint satisfaction problem.**

I. INTRODUCTION

A. Introduction to Benchmarking

From a general point of view, the goal of benchmarking practices is to compare alternative elements or processes through a performance rating based on well defined metrics. In the field of Information Technologies and Networks, benchmarks aims at guiding the selection of different implementations and configurations of software and hardware from a performance viewpoint. For instance, one may be interested in comparing the performance of a number of Java Virtual Machines (JVM), databases, application servers, etc. Benchmarks are basically specifications, edited by organizations like the Standard Performance Evaluation Corporation [1] or the Transaction Processing Performance Council [2]. Some benchmarks, like RUBiS [3][4] in the field of web application servers, also publish a reference implementation.

Benchmarking a client-server system typically takes:

- a reference application that uses the scoped alternative elements;
- a workload specification defining the flow of requests submitted to the reference application;
- a set of performance metrics of interest (e.g., application response times or computing resources usage).

For example, RUBiS provides an on-line auction web application, implemented with different design patterns, that can be run on a variety of JVMs, applications servers and databases. RUBiS also provides an HTTP traffic injection utility that defines a special mix of different HTTP requests. By measuring the performance of this application with different elements, the tester can compare performance of these alternatives and make the best technological choices.

B. Benchmarking and Optimization

One of the key issues with benchmarking is to get meaningful, actually comparable measures. As a consequence, all alternatives must be tuned, or, in other words, optimally parametrized to reach their best performance. Then, the benchmark is used to rate and compare possible parametrizations and find the optimal set of parameters values for the whole set of involved elements.

To go on with the RUBiS example, JVMs, applications servers and databases all have specific parameters whose settings may (or may not) influence the overall application performance. Let us mention heap size, garbage collection policy, size of thread pools, size of database connection pools, size of caches, etc. For example, [5] provides a tuning guide for Java EE applications, giving the main parameters of interest for performance. Of course, optimal settings of one alternative are likely to depend on the application. They also depend on the other elements' settings in the whole system.

To conclude, benchmarking activities typically relies on looping on the following steps:

- setting the system elements' parameters values to improve the system performance (tuning),
- benchmarking the system to rate the performance of this new parametrization.

Our idea is to develop an autonomic computing approach to benchmarking [6]. The full vision encompasses both the benchmarking step and the tuning step. This combination comes with two major interests: automation of benchmarking campaigns, and pre-optimization of a system before its actual production use.

C. Self-Regulated Load Injection

For the autonomic benchmarking step, we reuse previous work featuring Self-Regulated Load Injection (SRLI), as published in [7]. SRLI consists in an autonomous exploration of the system performance under a growing load injection. SRLI is based on a control loop enabling a smart workload increase according to the evolution of relevant metrics. SRLI stops as soon as some given saturation criteria, such as service response time or server's CPU usage, passes given thresholds.

 ISBN: 978-1-61208-134-2

As common load injection tools, SRLI defines the workload level in terms of number of active virtual users. One virtual user represents an elementary sequence of requests and think times, typically representing a real user session. Starting from a single virtual user, SRLI adds virtual users until reaching a saturation threshold. Finally, SRLI delivers a number of metrics such as average response time or server throughput. But the most relevant metric for our optimization purpose is the maximum number of virtual users that could be run before reaching saturation. As a matter of fact, it meters the maximum server's capacity under given operating and quality of service constraints.

On the software architecture side, SRLI is built as a component-based system, according to the Fractal model [8]. It springs from research work aiming at providing a component-based framework for building autonomic systems [9], as well as from the CLIF component-based load testing framework [10].

D. Introducing self-optimization

This paper is mainly dedicated to self-optimization aspects. It also sketches its integration with SRLI to provide the full self-benchmarking vision, keeping the component-based architectural design. Our approach to self-optimization is based on a classical generate and evaluate process: choose a set of parameter values and apply them to the system under test, and then, evaluate the performance resulting from these new settings, with SRLI. This actually introduces a new control loop that we call the optimization loop.

This approach must cope with a number of issues:

- gracefully handle multiple autonomic control loops;
- possible constraints between parameters' values;
- the huge combinatorial effect between the possible values of all the parameters;
- duration of each SRLI run, which is typically several minutes for our sample web application;
- automation of parameter values change.

E. A constraint solving approach

Besides this optimization control loop principle, the peculiarity of this work is to represent the optimization problem as a Constraint Satisfaction Problem (CSP) [11]. CSP is a convenient support for representing parameters' possible values and possible constraints between parameters. It is also a convenient way of generating valid parametrizations and submit them to SRLI's evaluation.

F. Paper outline

This paper is organized as follows: section II shows how we introduce the optimization control loop and combine it with SRLI. In Section III, we detail the algorithmic aspects of our self-optimization approach, including tools, strategies and heuristics. Section IV presents our first experimental results with a Java EE application use case. Finally, we conclude in Section V, and give some open questions and perspectives.

II. Introducing Self-Optimization

A. A component-based approach

Autonomic computing [12] springs from the observation that today's Information Technologies and Networks (IT&N) systems have reached such a complexity level, that their management reaches the limits of human capabilities. In a way, autonomic computing consists in using part of the IT&N power to handle its highly complex management. Now, a trap would be to fight complexity by adding even more complexity.

As mentioned in section I, SRLI has been designed according to a general component-based approach to building autonomic systems [9]. This approach advocates for a strong architectural requirement, to cope with this possible paradox of autonomic computing. The challenge is to limit and overcome the complexity of autonomic features, and keep a full control on them. The approach consists in having a uniform component-based system representation. Autonomic control loops are themselves built as components, which provides a comprehensive and self-aware architecture.

B. Architecture of Self-Regulated Load Injection

The initial idea behind SRLI is to consider that benchmarking activities are relevant use cases for this architectural approach to building autonomic systems, because of the high complexity level of load testing infrastructures. SRLI's goal is to automatically and quickly find the performance limits of an arbitrary system. [7] shows its practical use for testing a multi-tier Java EE application, and finding its performance limits in less than 10 minutes.

SRLI's top level components are:

- *load injectors*, in charge of generating workload on the tested system, and measuring requests response times and throughput;
- *probes*, responsible for monitoring usage of computing resources (processor, memory, network…);
- one *supervisor*, giving a central access point to control and monitor all probes and load injectors;
- one *load controller*, adjusting the injected workload, in terms of number of virtual users, according to performance measures;
- one *saturation controller* checking whether saturation criteria are met or not.

SRLI combines two control loops:

1) the load injection control loop, adjusting the number of virtual users according to the response times and throughput observations. It involves the load injectors, the supervisor, and the load controller;
2) the saturation control loop, in charge of stopping the load injection control loop when some saturation thresholds are reached. It involves the probes, the supervisor, and the saturation controller.

Both loops have explicit control on each other: loop 1 first launches loop 2, and then loop 2 stops loop 1. Loop 1 is launched by external control e.g., by a user, to restart the process on a new system configuration.

 ISBN: 978-1-61208-134-2

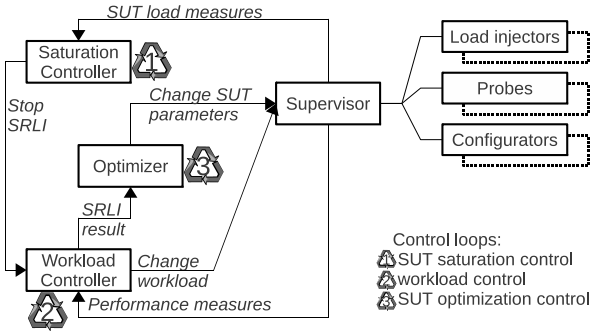

Figure 1. Global self-optimization architecture with three control loops

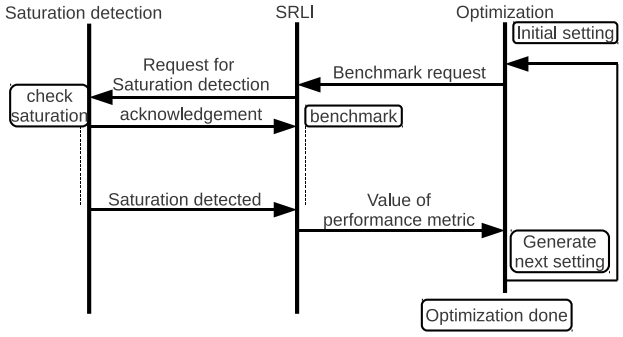

Figure 2. Sequence diagram of full autonomic benchmarking

C. Introducing our self-optimization components

Our self-optimization control loop is built from the following components:

- an *optimizer* component in charge of generating a new parametrization according to the performance metrics values it has gathered during the previous SRLI completions;
- *configurator* components in charge of applying the new parameters' values to the benchmarked system.

Section III gives details about an optimizer component based on constraint solving paradigm. Configurator components must typically stop the system under test, apply the new parametrization, and restart the system. Parameters may be set through some management protocols, such as SNMP [13] or JMX [14], or by updating configuration files. Figure 1 shows the global architecture combining SRLI and self-optimization components. The logical sequence between the three control loops' activities is depicted by figure 2.

D. Focus on control loops coordination

This global self-benchmarking approach is a use case for multiple control loops coordination. There exists three main schemes:

- explicit control between control loops, as it is currently implemented in SRLI: control loops are aware of each other;
- hierarchical control i.e., low-level control loops don't know each other but are controlled by a higher level control loop;
- implicit control i.e., control loops don't know each other but communicate through their environment.

In this work in progress, we currently use the straightforward explicit control scheme. However, we are interested in exploring the implicit control approach, in order to be able to combine some, but not necessary all, of these three control loops, possibly with other control loops. As a noteworthy example, we may be interested in keeping saturation detection and self-optimization, while replacing our massive load injection by a thin load injector checking quality of service. This way, we could reuse some of our components to support self-optimization in a production environment instead of a benchmarking environment.

III. Algorithmic aspects of Self optimization - A first attempt

In this section we explain how we find the values of the parameters required to configure the system under test to get the best behavior for a given metric m.

A. Coping with the complexity of system tuning

As mentioned in the introduction, there are a lot of parameters to set for tuning a system like an application running on a Java EE server. To manage these parameters, we can define two sets of parameters classes. The first set is related to the *software level*, that is in what part of the hardware/software stack is located the parameter. We can distinguish four classes: OS and hardware parameters, JVM parameters (memory, size of the heap, ...), Java EE or other server parameters (if this applies), Application parameters. Another set of parameters is related to the *technical level*, that is when (with regard to the running state) we can/must change the value of the parameter. We have defined three classes: before deployment of the application, before starting the application server (if this applies), when the application runs. Note that, to modify the value of a parameter, several technical means may be used such as files and JMX interactions.

Another classification of the parameters relies on the metric m. To devise efficient algorithms, it is necessary to establish how m varies with each parameter p_i, for fixed values of the other parameters. Let us denote by $\bar{p} = (p_1, \ldots, p_n) \in D = \prod_{i=1}^{n} D_i$ the vector of parameters we control for tuning the system. Based on our previous works on self-benchmarking, we assume that the function $f(\bar{p}) = m(\bar{p})$ is concave on D. So, if we denote by f_i the "projection" functions of f on D_i, that is: $f_i(p_i) = f(\bar{p})$ for a given $(n-1)$tuple of parameters $p_j (j \neq i)$, we know that f_i has a unique maximum (at $p_m \in D_i$). This specific property of the metric m allows us to devise an adapted algorithm. The second classification of the parameters gives the relative impact of each parameter with regard to m. These information are provided by human experts and from models of the system studied if they are available.

The above classes of parameters allows us to restrict the set of parameters to n parameters $p_1, p_2, \ldots, p_n$, each one with a domain $D_i = [min_i, \ldots, max_i]$. Note that we can

restrict each D_i to a finite set of integers. In our example (see Section IV), we have found at least 400 parameters that we can control. From these, we selected 15 significant parameters and in this paper, we present first results with two parameters.

B. Finding the best parameters

The problem is then to find $p^{opt} = argmax_{\bar{p} \in D}\{m(\bar{p})\}$. The main difficulty to find p^{opt} is that the computation of $m(\bar{p})$, which corresponds to a self-benchmarking experiment takes several minutes. Hence, we have to devise an algorithm requiring as few as possible such computations.

To reduce the number of calls to m, we first use the context of Constraint Satisfaction Programming (CSP) (see [11] for a presentation of CSP). The main interests of CSP for our problem is the ability to define parameters, their domains and their constraints in a declarative way and to check at runtime if a given parameter tuple is an admissible solution or not. Among the available solver tools, we have chosen the Choco [15] open source library. Choco provides several Java libraries to define the CSP and to verify the constraints. Obviously, Choco embeds also a customizable solver, but in this paper we did not use it, but in contrast we use our specific traversal algorithm in the domain D.

Algorithm 1 is based on a kind of binary search method. For each parameter p_i, we define a granularity g_i which corresponds to the minimal distance we require between two values of p_i. Then, we define the norm of a vector $\bar{p}$ as $||\bar{p}|| = \sum_{i=1}^{n} |\frac{p_i}{g_i}|$, so that we can compute the relative variation of $\bar{p}$ between $\bar{p}_c$ and $\bar{p}_p$ as $var(\bar{p}) = \frac{||\bar{p}_c - \bar{p}_p||}{||\bar{p}_p||}$. We also set the precision x of the search and the maximal number k_{max} of the main iteration. Each iteration of the algorithm computes the vector $\bar{p}_c^{opt}$, starting from its first component, and updating successively its n components. For a given component i, the function findbest computes the value of p_i which maximizes $m(\bar{p}|i, x) = m((p_1, \ldots, p_{i-1}, x, p_{i+1}, \ldots, p_n))$ findbest selects the values to be used to compute the metric m in a "dichotomous way", with the idea that, based on previous computations of m, we can skip some new computations. This approach gave interesting results during our first experiments (see Section IV).

Algorithm 1 Compute the optimal parameters

```
Define the initial value of p_c^opt
p_p^opt = (1 + 2y) p_c^opt
k = 1
while (var(p^opt) > y) and (k ≤ k_max) do
  for i = 1 to n do
    (p, m) = findbest(i, p_c^opt, min_i, max_i)
    p_{c,i}^opt = p
  end for
  k = k + 1
end while
return p_c^opt

Function (int, double) findbest(i, p̄, min, max)
while (max − min ≥ g_i) do
  mid = (min + max)/2
  compute m_{i,min} = m(p̄|i, min)
  compute m_{i,mid} = m(p̄|i, mid)
  if m_{i,min} > m_{i,mid} then
    return findbest(i, p̄, min, mid)
  else
    compute m_{i,max} = m(p̄|i, max)
    if m_{i,mid} < m_{i,max} then
      return findbest(i, p̄, mid, max)
    else
      v_1 = findbest(i, p̄, min, mid)
      compute m_{i,v_1} = m(p̄|i, v_1)
      if m_{i,v_1} < m_{i,mid} then
        v_2 = findbest(i, p̄, mid, max)
        compute m_{i,v_2} = m(p̄|i, v_2)
        if m_{i,v_1} > m_{i,v_2} then
          return (v_1, m_{i,v_1})
        else
          return (v_2, m_{i,v_2})
        end if
      else
        return (v_1, m_{i,v_1})
      end if
    end if
  end if
end while
return (mid, m_{i,mid})
```

IV. First results with a Java EE application server

A. Use case: JOnAS 5

JOnAS [16] is an open source, Java EE 5 certified, application server provided by the OW2 consortium. JOnAS is based on an OSGi framework, which provides mechanisms to dynamically change bundles' configuration (start, stop, reconfigure, etc.). All JOnAS components are packaged as bundles and the full JOnAS profile comes with more than 250 bundles. Therefore most Java EE-certified JOnAS services (persistence, EJB, resources ...) are available as OSGi services to all OSGi bundles deployed on JOnAS.

JOnAS administration is performed through Java JMX [14], either with graphical tools such as JOnAS' web console (JonasAdmin) or common Java's jconsole utility, or with a command line tool like MBeanCmd provided by the JASMINe open source project [17].

For our experiment, we need to stress a benchmark application. We have chosen to deploy the simple e-commerce web application MyStore 2.0.2 on JOnAS. This sample application is available from the JASMINe project repository. This is a very simple application that does not use a database.

B. Tuning points

Parameters may be set in configuration files or through JMX if they are available as MBean attributes. In the first case, changing a parameter value requires to modify a configuration file, and most probably restart JOnAS or at least one of its services. In the latter case, a simple JMX call to JOnAS' embedded JMX server allows for changing a parameter value.

JOnAS 5 comes with 25 services and 48 configuration files, resulting in one hundred parameters. A parameter definition may set a simple type value such as an integer or a Boolean, or a complex value such as a policy definition (cache management, garbage collector ...). In the case of a policy change, the system behavior may completely differ. So, the algorithm described in section III should be fully applied, consecutively for each policy setting. As a matter of fact, the concavity assumption can't hold with such parameter types. Note that Boolean values may be regarded either as integer values or policy definition, for they have only 2 possible values that must be both evaluated anyway.

A tuning point is a parameter which impacts "heavily" the server performance. Of course, there is no formal definition of "heavily". However, some studies (see for instance [5]) in the performance testing and optimization field show that we can grab quickly 80% of performance improvement tuning the JVM heap, the thread pools, the connection pools and the caches. The remaining 20% can be obtained tuning the EJB pools, the JMS and pre-compiling JSPs. So, based on experts' works, we can define a set of main tuning parameters.

For our experiments with MyStore, we have considered the two following tuning points:

- the size of the thread pool of the HTTP connector. It is controlled by its maximal value, noted `maxThreads`.
- the size of the Application Cache. It is also controlled by its maximal value, noted `cacheMaxSize`.

To determine the value range of parameter `maxThreads`, we did an initial experiment with the default value 200. We monitored the MBean attribute `currentThreadsCount` to observe the actual number of server's active threads. From the observed values, we decided to set `maxThreads`'s range to 20 – 200. We chose a granularity of 20 threads, as a trade-off between optimization accuracy and expected experimental time. If we applied an exhaustive exploration of this range with values obtained with the same dichotomous principle as described in section III, we would get 9 possible values.

We applied the same protocol to determine `CacheMaxSize`'s range. By default it is initialized to 10240KB. In fact, since MyStore does not use so much the cache (only a few images of reduced size are cached), we expect this value could be substantially lower. We monitored the MBean attribute `cacheSize` and found out that range 10 – 100 would be relevant. We chose a 10KB granularity. There again, an exhaustive exploration of this range with values obtained with our dichotomous principle would give 9 possible values.

Note that, even if it took some time to evaluate the boundaries of the parameters, this step allowed us to reduce the size of the couple space to be searched for optimal values.

Table I
EXPERIMENTS RESULTS

k	i	(p1,p2)	m	duration	max-min	results
1	1	(10,20)	177	7min25s	ok	
1	1	(55,20)	144	5mn15s	ok	
1	1	(32,20)	169	5mn30s	ok	
1	1	(21,20)	133	5mn20s	no	$p_1^{opt1} = 10$
1	2	(10,110)	131	5mn45s	ok	
1	2	(10,65)	120	5mn0s	ok	
1	2	(10,42)	177	7mn25s	ok	$p_2^{opt1} = 42$
2	1	(55,42)	159	5mn55s	ok	
2	1	(32,42)	184	8mn31s	ok	
2	1	(21,42)	144	5mn25s	no	$p_1^{opt2} = 32$
2	2	(32,110)	151	6mn25s	ok	
2	2	(32,65)	144	5mn46s	ok	
2	2	(32,42)	184	8mn31s	no	$p_2^{opt2} = 42$

C. Experiments and results

We use SRLI to generate an increasing load on MyStore. Each virtual user runs a simple scenario resulting from a real user session capture, enriched with random think times. SRLI's stopping criteria are defined as follows:

- the SUT's CPU usage must be less than 70%,
- the SUT's JVM heap memory usage must be less than 95%,
- the SUT's RAM usage must be less than 80%.

From a technical viewpoint, the experiment infrastructure uses three computers on a Gbit/s Ethernet network:

- the SUT runs on a server with two 2.8 GHz Xeon processors and 2 GB of RAM. CPU, JVM, network and memory probes are deployed on this server;
- load injection is run on one server with a quad-core 2.0 GHz Xeon processor and 8 GB of RAM;
- the supervisor and controller components (see figure 1) are run on a common PC, whose properties do not matter for the test.

Table I shows the results obtained. k and i denote respectively the number of the main (while) iteration in Algorithm 1 and the index (the number of the parameter) in the for loop; (p_1, p_2) gives the values of the two parameters. m reports the number of virtual users, while duration is the time required to get m. "ok" (resp. "no") in the max-min column indicates if the granularity was (resp. was not) reached during the test for the given parameter couple. We note that our algorithm provides interesting savings due to the reduced number of tests with respect to the number of couples; we actually reduce the number of saturation tests from 81 to 12. This reduction springs from the concavity assumption.

 ISBN: 978-1-61208-134-2

V. Conclusion and future work

Benchmarking requires tested systems to be optimized, to get meaningful results. This paper presents a global vision of autonomic benchmarking addressing this issue. This vision combines an autonomous system for measuring the maximum capacity of a system, given some operating and user experience constraints, with a self-optimization feature. We describe the global architecture, based on three component-based autonomic control loops, which explicitly coordinate with each other.

Then, the paper focuses on work in progress on self-optimization. The basic principle is to generate valid settings of parameters for the tested system, apply these settings, and to get a performance rating from the autonomic load testing loop. To practically enable this idea, we address a number of issues:

- representing domains of parameters' values, including possible constraints between one another. We propose to use Constraint Satisfaction Programming.
- limiting the huge combinatorial effect between parameters values (algorithmic complexity) and the load testing time. Assuming that the metric, as a function of all parameters, has no local maximum value, our algorithm drastically limits complexity.

Our first experimental results on a sample Java EE web application are promising. We actually needed to benchmark only 12 out of 81 valid settings, each benchmark duration being limited to an average 7min30s. Hence the global campaign lasted less than 2 hours instead of about 10 hours and 40 minutes (taking parameters setting time and system restart duration).

Our future work will be dedicated to complete our optimization algorithm implementation, and to actually integrate benchmarking and optimization, to get a fully autonomous system. We will build the global architecture with an implicit coordination between control loops, to get loosely coupled controller components.

Acknowledgment

We thank the Choco team and the JOnAS team for their valuable support. This research is supported by the French ANRT and Agence Nationale de la Recherche with the ANR-08-SEGI-017.

References

[1] "Standard performance evaluation corporation," http://www.spec.org/, checked on 18th March 2011.

[2] "Transaction processing performance council," http://www.tpc.org, checked on 18th March 2011.

[3] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance comparison of middleware architectures for generating dynamic web content," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, ser. Middleware '03. Springer-Verlag New York, Inc., 2003, pp. 242–261.

[4] "RUBiS - Rice University Bidding System," http://rubis.ow2.org/. Checked on 18th March 2011, Nov 2009.

[5] S. Haines, *Pro Java EE 5, Performance Management and Optimization*. Apress, 2006.

[6] F. Boyer, C. Taton, J. Philippe, and B. Dillenseger, "Selfware self-optimization: algorithms, architecture and design principles," Selfware Deliverable SP2-L2, http://sardes.inrialpes.fr/~boyer/selfware/documents/SP2-L2-Auto-Optimisation.pdf, Tech. Rep., june 2008, checked on 7th April 2011.

[7] A. Harbaoui, N. Salmi, B. Dillenseger, and J. Vincent, "Introducing queuing network-based performance awareness in autonomic systems," in *Proc. 2010 Sixth International Conference on Autonomic and Autonomous Systems*, ser. ICAS '10. IEEE Computer Society, 2010, pp. 7–12.

[8] G. S. Blair, T. Coupaye, and J.-B. Stefani, "Component-based architecture: the fractal initiative," *Annales des Télécommunications*, vol. 64, no. 1-2, pp. 1–4, 2009.

[9] ANR Selfware project, "Selfware: Lessons learned to build autonomic systems," http://sardes.inrialpes.fr/~boyer/selfware/documents/SP1-L3-Architecture.pdf. Checked on 7th April 2011, 2008.

[10] B. Dillenseger, "Clif, a framework based on fractal for flexible, distributed load testing," *Annales des Télécommunications*, vol. 64, no. 1-2, pp. 101–120, 2009.

[11] F. Rossi, P. van Beek, and T. Walsh, *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. New York, NY, USA: Elsevier Science Inc., 2006.

[12] J. O.Kephart and D. M.Chess, "The vision of autonomic computing," IBM Thomas J.Watson Research Center, january 2003.

[13] IETF, "A simple network management protocol (rfc 1157)," http://www.ietf.org/rfc/rfc1157.txt. Checked on 18th March 2011, May 1990.

[14] J. C. Process, "Java management extensions (jsr 3)," http://www.jcp.org/en/jsr/detail?id=3. Checked on 18th March 2011, November 2006.

[15] H. Cambazard, N. Jussien, F. Laburthe, and G. Rochart., "The choco constraint solver." in *INFORMS Annual meeting, Pittsburgh, PA, USA*, Pittsburgh, PA, USA, November 2006.

[16] OW2.org Consortium, "JOnAS Java open application server," http://jonas.ow2.org. Checked on 7th April 2011.

[17] OW2 Consortium, "JASMINe, the smart tool for your SOA platform management," http://jasmine.ow2.org. Checked on 7th April 2011.

 ISBN: 978-1-61208-134-2