Journal of Computer Science 2 (6): 513-520, 2006
ISSN 1549-3636

Predictive Autonomicity of Web Services in the MAWeS Framework

[1]Emilio P. Mancini, [2]Massimiliano Rak, [2]Roberto Torella and [1]Umberto Villano
[1]RCOST and Dipartimento di Ingegneria, Università del Sannio, Italy
[2]Dipartimento di Ingegneria dell'Informazione, Seconda Università di Napoli

Abstract: In Web Services designs classical optimization techniques are not applicable. A possible solution to guarantee critical requirements is the use of an autonomic architecture, able to auto-configure and to auto-tune. This study presents MAWeS (MetaPL/HeSSE Autonomic Web Services), a framework whose aim is to support the development of self-optimizing predictive autonomic systems for Web service architectures. It adopts a simulation-based methodology, which allows to predict system performance in different status and load conditions. The predicted results are used for a feedforward control of the system, which self-tunes before the new conditions and the subsequent performance losses are actually observed.

Key words: Autonomic, self-optimization, web services, performance prediction, simulation

INTRODUCTION

The use of Web Services architectures is becoming a customary approach for the development of open, large-scale interoperable systems[1–5], and there are many examples of "working" solutions. But reliability, availability, performance and security of these architectures are completely open issues. In Web Services designs, due to architecture transparency, classical techniques for system optimization (such as *ad-hoc* tuning, performance engineered software development, ...) are not applicable. In practice, the only solution to guarantee critical requirements seems to be the use of an architecture able to auto-configure and to auto-tune, until the given requirements are met.

Autonomic computing[6-9] whose name derives from the autonomic nervous system, aims to bring automated self-management capabilities into computing systems. Autonomic capabilities are usually classified as:

* **Self-configuring** the system can dynamically adapt to changing environments;
* **Self-healing** the system can discover, diagnose and react to disruptions;
* **Self-optimizing** the system can monitor and tune resources automatically;
* **Self-protecting** the system can anticipate, detect, identify and protect itself against threats.

Even if currently there are many simple examples of application of autonomic concepts, and a toolkit for building autonomic systems is available[10,11], all known solutions are fairly "young" and rather unstable. The majority of these solutions are based on *reactive autonomicity*, or, stated another way, on feedback control. The continuous analysis of system logs and/or direct monitoring point out configuration or performance problems. The system automatically reacts applying new configurations or tuning policies. Recently a new approach[12] based on *predictive autonomicity*, i.e., on feedforward control, has been proposed. In this case, the system detects and forecasts variations in parameters and their impact on performance, and self-tunes, anticipating the need.

In this study we describe MAWeS (MetaPL/HeSSE Autonomic Web Services), a framework whose aim is to support the development of self-optimizing predictive autonomic systems for Web Services architectures. It adopts a simulation-based methodology, which allows to predict system performances in different status and load conditions. The predicted results are used for a feedforward control of the system, which self-tunes *before* the new conditions and the subsequent performance losses are actually observed.

METAPL/HESSE METHODOLOGY

HeSSE (*He*terogeneous *S*ystem *S*imulation *E*nvironment) is a simulation environment that allows the user to simulate the performance behavior of a wide range of distributed systems for a given application, under different computing and network load conditions.

The HeSSE compositional modeling approach makes it possible to describe Distributed Heterogeneous Systems by interconnecting simple *components*. Each component reproduces the performance behavior of a section of the complete system at a given level of detail. A HeSSE component is basically an object, hard-coded with the performance behavior of a section of the whole system. More detailed, each component has to reproduce both the functional and temporal behavior of the subsystem it represents. In HeSSE, the functional behavior of a component is the service set exported to

Corresponding Author : Umberto Villano, Università del Sannio

the other components. So, connected components can ask other components for services. The temporal behavior of a component describes the time spent servicing.

HeSSE uses traces to describe applications. When the application is not available, e.g., it is still being developed, they can be generated using prototypal languages. In the past years, we defined an XML-based meta-language for parallel programs description, MetaPL[13,14]. It is language independent, and can support different programming paradigms or communication libraries. The core MetaPL notation can be extended through Language Extensions (XML DTDS), which introduce new constructs into the language. Starting from a MetaPL program description, a set of extensible filters makes it possible to produce different program *views*, among which are the trace files that can be used to feed the HeSSE simulation engine. Among available MetaPL extensions, are ones that enable the description of remote services interfaces, client calls to remote services and stub creation. The detailed description of the MetaPL approach to program description, and of the trace generation process, is out of the scope of this study and can be found in references 14 and 15.

MAWeS FRAMEWORK

HeSSE and MetaPL allow users to perform system performance analysis without actual execution of the application on the target environment. In previous studies, we have presented the whole development procedure, the validation of the results and their analysis, showing pros and cons of the approach[13-17]. In this study we tackle a new problem, i.e., how to automatize the simulation and performance prediction process, in order to develop self-optimization autonomic systems for Web Services architectures.

In the systems which are the object of this study, i.e., in distributed systems running Web Services applications, it is possible to exploit self-optimization techniques at three different, non exclusive, levels:

* **Server Level** the autonomic system affects the underlying distributed system, i.e., the optimization actions are performed at the operating system and network levels;
* **Service Provider Level** the autonomic system affects the service provider, i.e., the optimization actions have impact on the tuning of parameters and on the workload management policies of each offered service;
* **Application level** the autonomic system affects user applications, modifying their resource use and the order of the actions they perform.

The stress in this study will be on the use of self-optimization techniques at application level.

As mentioned before, the solution we have devised for the development of autonomic Web Services applications hinges on the use of the MAWeS framework, which (partially) hides the presence of a simulation environment exploited through a web service interface.

The MAWeS framework (Fig. 1) is structured in three layers, as follows:

* **Frontend** made up of the software modules used by final users to access the MAWeS services;
* **Core** composed of the software and the services that manage MetaPL files and make optimization decisions;
* **WS interface** the set of Web Services used to obtain simulations and predictions through MetaPL and HeSSE.

In its current implementation, the MAWeS Frontend provides a standard client application interface, `MHAWeSclient`, which has to be extended by developers with their actual application code. The MHAWeSclient client accepts as input a MetaPL file describing the application code. It should be explicitly noted that the final objective of our research is automatic generation of application code from a skeletal MetaPL description. However, currently the writing of application code from its high-level MetaPL description is still performed manually.

The MAWeS Core exploits environment services (i.e. the services offered by the environment to monitor and to manage itself) and the MetaPL/HeSSE Web Services interface using the application information contained in the MetaPL description to find out optimal execution conditions. It is a software unit provided both as a web service and integrated into the `MHAWeSclient`.

The MetaPL/HeSSE WS interface, thoroughly described in the following section, defines a set of services that make it possible to automatize the methodology application.

The sequence of events and calls that allow the execution of an application with optimal values of parameters is also shown in Fig. 1.

The MAWeS client submits the MetaPL application description to the MAWeS core (1). Then the services of the framework automatically find out the set of simulations needed, perform them (2,3,4), and return the set of optimal parameters for the target application (5). Finally, `MHAWeSclient` starts the application code, passing to it the set of optimized parameters (6).

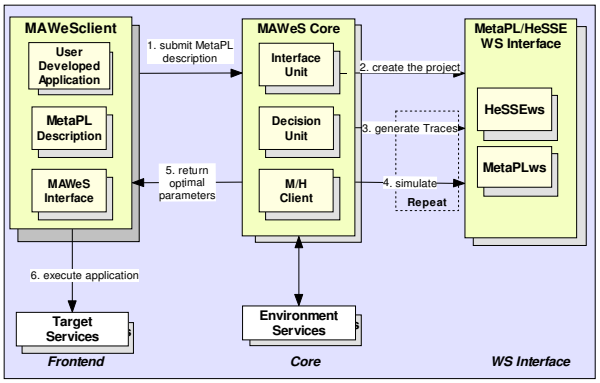

Fig. 1: The MAWeSclient and the MAWeS services

The critical point of the whole autonomic optimization procedure is to point out the set of parameters that mostly affect performance, in order to find automatically the best application configuration. New MetaPL extensions have been developed for this purpose. These will be described in the following. The following sections will describe in detail each of the three layers of the framework, starting from the lowest level services (WS interface), to the highest level of abstraction (Frontend).

METAPL AND HESSE WEB SERVICES INTERFACE

MetaPL descriptions and HeSSE simulation make it possible for system and program developers to identify performance bottlenecks and to tune both the system and the final applications. Even if these tools can be used through a command-line or graphical interface[17], we have developed a set of Web Services that allow the user to perform the description, simulation and analysis steps. This Web Service interface can be exploited by developers, who want to access the simulator and to perform analysis manually, or by software tools such as the MAWeS core to automatize the performance analysis process. In the following, we will briefly describe the newly developed Web Services interface.

HeSSEws: The input for HeSSE simulations is a set of files, namely, configuration and trace files. The simulator outputs log files containing all the events and timings of the simulated model. Unlike in previous versions of the simulator, currently the logs are produced in XML format, which can be more easily managed through automatic tools. The HeSSE Web Service `HeSSEws` gives a good level of control over the simulator through a number of methods that make it possible to make a new directory into the Web Service base folder and to create a file (`mkDir`, `createFile`), to start the simulation using the contents of a folder and to read the simulator output (`runSimulation`, `readOutput`).

MetaPLws: MetaPL operates on a set of files, namely MetaPL documents, schemas, filters and final views (simulation traces are a particular type of view). Therefore the MetaPL Web Service interface components offer a set of services able to store, modify and generate these documents. MetaPLws publishes the Web Services `applyFilter`, which applies a filter to a MetaPL document, and `execute`, which executes a script (typically applying a sequence of filters). These make it possible to implement the methodology outlined in the previous Section.

The system stores internally the set of DTDs, Schemas, and XSLT files needed to validate or to transform a MetaPL file, but it can also use public or private (submitted by the user) repositories. The `applyFilter` service accepts as input the MetaPL file and the name of the filter to apply, and returns the views generated by the filter.

It is often necessary to apply multiple filters, thus generating multiple output files. In these cases, it is useful to resort to the `execute` service. This service accepts as input a file attached to the SOAP message that contains a compressed folder. The service extracts the attachment, and gives the result to the software unit that manages the script files (`MetaFilter`), which interprets the script describing the actions to be performed and executes them.

THE MAWES CORE

The MAWeS Core is invoked by the MAWeS-client, as previously shown in Fig. 1, in a way completely transparent to the application. The core contains all the autonomic "intelligence" of the framework, in that it has the task to find the optimal parameters for application execution. Each time that the client calls the service, the MAWeS Core finds which simulations are needed, generates all the traces required for simulation and performs the simulations. Then, it analyzes the results and chooses the optimal parameter configuration.

The MAWeS Core is structured in three modules, as shown in Fig. 2.

* **Interface** this unit has the task to recover all the information on the target application needed for optimization. In particular, it extracts from the MetaPL description of the application the MetaPL autonomic extension tags and passes them to the Decision unit;
* **Decision** this unit contains the intelligence of the autonomic environment. It chooses the possible values to evaluate, predicts the (future) system status, defines the optimal optimization criteria, queries the simulation engine through the M/H client, and makes the final decisions;
* **M/H client** this unit executes the simulations and the analysis defined by the decision unit, using the MetaPL/HeSSE Web Service interface.

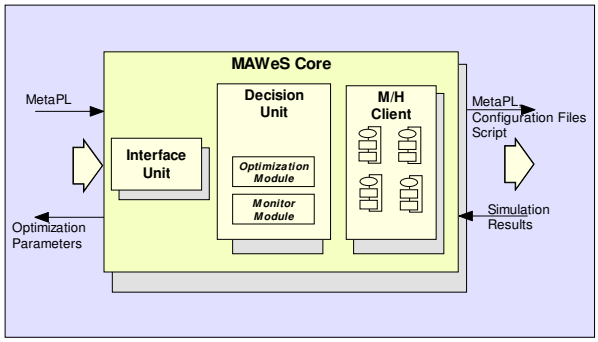

Fig. 2: The MAWeS Core structure

These modules are in turn described below.

The interface unit: The task of the Interface Unit is to find all the parameters that may affect the application performance and to pass this information to the Decision Unit. In order to do so, it extracts from the MetaPL application description all information that may possibly be used to optimize the application. At least in theory, optimization parameters could be automatically extracted from MetaPL application description. However, in the current implementation of the framework, the user has to choose them manually.

The Web Services MetaPL extensions allow the description of service-oriented applications. Further metalanguage extensions are needed to automatize the self-optimization steps. MetaPL descriptions consist essentially of code prototypes, enriched with task-to-processor mapping (`Mapping` tag). The Autonomic MetaPL extensions define new language elements for this section. They introduce the `Autonomic` tag, included in `Mapping` element, which describes the target simulation configurations that can be used for application execution.

As mentioned before, the Autonomic MetaPL extensions provide new tags to be included in the `Autonomic` element:

* **`Parameter`** identifies a variable inside the description that affects the overall performance, and thus one whose optimal value has to be evaluated by the framework. The (optional) attributes `min` and `max` represent the minimum and maximum values the variable can assume. In alternative, it is possible to insert into the `Parameter` element content the list of the values that the parameter can assume.
* **`Target`** identifies the output parameter adopted for system optimization. In the MAWeS current implementation, the only possible optimization target is response time minimization, and hence the only target of optimization is a minimization rule. Future versions of the framework will also provide an attribute to express alternative objective functions.

The Core interface exposes its services to final user as Web Services. Future implementations of the framework may integrate it directly into the MAWeS Frontend, moving the overhead from the server to the client in order to reduce communication costs. In fact, for the time being, the Core offers only a single service, `MAWeSoptimization`, which starts the application performance optimization procedure. The interface receives the MetaPL description and extracts all the information contained in the autonomic extensions, then calls the MAWeS decision unit with the description and the optimization data as parameters.

The decision unit: As mentioned before, the Decision Unit contains all the framework autonomic intelligence, and applies the optimization rules defined by the framework administrator to optimize the target applications by means of the feedforward approach described in the introduction. More detailed, whenever the Decision Unit has to make an optimization decision (typical examples are when an application is launched, or at periodic intervals in time), it queries the underlying target system through a *Monitoring Module*. This is interfaced to a distributed monitoring subsystem, and returns the current system load (e.g., CPU load of the component nodes, current network traffic, ...). This information is used by the Decision Unit to predict the *future* system load (in the two mentioned examples, during the following application run, or during the next time interval, respectively). The description of the method used to foresee the load in a future interval in time is out of the scope of this study. Roughly speaking, it is possible to exploit previous knowledge to do so (e.g., in systems where there is a typical load profile in time), or to make predictions based on the current trend of system load. Whatever the method used, the Decision Unit optimizes the system *before* the new load condition is observed, performing a set of simulations (through the M/H client) to find the best operating condition *in the predicted load status*. The optimization criteria used are applied by a suitable sub-unit, the Optimization Module.

Independently of the load prediction method adopted and of optimization criteria chosen, the unit always works as follows:

* it obtains from the MAWeS Interface the application MetaPL description and the list of parameters to be optimized;
* it builds the application MetaPL description and the simulator configuration files;
* it builds a set of metafilter scripts;
* it starts as many M/H clients as the number of different simulations required;
* it collects all the simulation results, analyzes them and finds the optimal parameter values.

It should be explicitly noted that in order to build the configuration files it is necessary to exploit the unit intelligence, as described above. Stated another way, finding the "useful" simulations to be performed

requires the future load to be predicted, and the optimization logic to be applied. All the simulations the M/H module is asked for, are independent of each other. Hence they can be executed as services on one, or even on multiple MAWeS service providers.

The modular structure of the Decision Unit, and the use of a separate Optimization Module, promotes the integration of new optimization criteria in the tool. For simplicity's sake, in the exposition that follows we will present a trivial optimization algorithm, based on full factorial exploration of all the possible configurations. In particular, we assume that each optimization parameter P_i can assume only a finite number of values N_i, and therefore a total of $\prod N_i$ distinct simulations are to be performed. As this is the most inefficient choice as far as the framework overhead is concerned (it requires the highest number of simulations before an optimization decision is made), it is worth pointing out explicitly that:

* Due to the use of this approach, the overhead figures that will be presented later can be considered (with some caveats) a higher bound for actual framework overheads;
* The factorial exploration of all possible configurations is limited to the set of values explicitly enumerated in the MetaPL description.

M/H client: The M/H (MetaPL/HeSSE) Client is a software unit that implements a client for the MetaPL/HeSSE web service interface. Its implementation is threaded, in that each client instance runs as a new thread. This makes it possible for the MAWeS core to start in parallel as many M/H clients as are necessary. Each client invokes the services needed to perform the simulation and analysis for each different configuration, and gets the corresponding results. The Decision Unit collects all the results, compares them and makes its decisions.

MAWeS FRONTEND

The current implementation of the MAWeS Frontend is composed of three classes, which should be extended by the final user, and enriched with the application MetaPL description. From the point of view of the application developer, the whole development process consists of the following steps:

* description in MetaPL of the application and of the application parameters;
* selection of needed services and of their service provider;
* generation of application code from the MetaPL description.

It should be noted once again that the parameters to be optimized and the optimization target are to be explicitly declared by the user. We aim to automatize this step in the future, developing filters able to extract this information from a MetaPL description.

Once the code has been developed, the developer just starts the application, which, in a completely automated way, calls the framework services. These evaluate optimal values of the parameters, and run the re-configured and optimized application on the target system.

MAWeS FRAMEWORK CASE STUDY

The proposed framework lets Web Services oriented application self-tune, without any intervention from the service provider or from the final user. The approach works independently of the presence of a local or global scheduler, even if it can take their presence into account. The main drawback linked to the presence of the autonomic framework is the introduced overhead. In cases where the framework is used to launch new instances of an application with optimized parameters adapting to the (expected) system load, as in the example proposed below, this leads to an increase of application startup time. This is due to the time spent in simulation and analysis before the (optimized) application instance can be executed.

In order to evaluate the framework performance, we present here a simple case study, based on a service-oriented tool for log file analysis. In our tests, we will measure the application response times, comparing the results obtained with and without the autonomic framework. Then we will also assess the impact of the choice of the feedforward approach for autonomic optimizations.

The LogAnalysis application: The Log Analysis application is a simple but realistic case study that shows how the framework affects the application execution performance. We have set up a web service provider, a simple set of services, and a WS-based application. The response time of the latter is the target of the framework optimizations.

The services exposed by the LogAnalysis server make it possible to perform Apache log file analysis. They are the following: `getLogLength`, which returns the log file dimension (in lines), `getLogFragment`, which accepts as parameters a size and a position into the log file, and returns a fragment of the log starting from the line given and of the given size, and `Unique`, which accepts as parameter a log file fragment and returns the IP addresses of the web clients logged in the fragment.

As regards the LogAnalysis client application, it periodically (every T seconds) evaluates the number of accesses of each web client (denoted by its IP address) scanning the log file by means of the above-described LogAnalysis services. The client can perform this

operation in a single step, thus interacting with only one LogAnalysis server, or in multiple steps (using the `getLogFragment` service and calling the `Unique` service on each fragment). In the second case, there can be a sharing of load between multiple LogAnalysis servers, all of which are provided with a copy of the same Apache log. As the client application is multithreaded, every fragment of the log file is processed concurrently by a different thread.

Every time that the log scansion is launched (i.e., every T seconds) the modulation of the number of LogAnalysis client threads, of the number of available LogAnalysis servers and the assignment of clients and servers, makes it possible to vary the load injected in the compute nodes and in the network, and hence the overall application response time. In the tests that will be presented below, we assume that the only CPU and network load present is the one generated (directly or indirectly) by the LogAnalysis application. However, the HeSSE simulation environment makes it possible to take into account the presence of additional load generated by other applications running concurrently with the software system under analysis. It is worth pointing out that studying the effect of these parameter modulations is not trivial, not to mention finding optimal values in every working condition (i.e., under different compute nodes and network loads).

The MetaPL description of the LogAnalysis client, not shown here for brevity's sake, explicitly points out the optimization variables, which are:

* **`NumberOfProcesses`** the number of LogAnalysis client threads. This parameter is linked to the dimension of the log file fragments (each fragment dimension is given by the total log file dimension divided by the number of clients);
* **`NumberOfServers`** the number of LogAnalysis servers;
* **`ServerChoicePolicy`** the LogAnalysis server choice policy adopted. Each client thread can ask any of the available servers for the services it needs. Among all possible policies, we only consider here a static and a dynamic one. Using the static policy, each client thread asks for services always to the same LogAnalysis server. The dynamic policy instead assumes that a client thread asks to different LogAnalysis servers.

The *LogAnalysisClient* execution is performed as follows. Every time that a log analysis scan is to be launched (every T seconds), the client asks the framework for the optimal number of client threads, of LogAnalysis servers and for the server choice policy (static or dynamic) to be used. The framework executes a set of preliminary simulations to find the optimal value for the described parameters *in the forthcoming environmental conditions* (compute node loads, network traffic), predicted for the next application run. The LogAnalysis client receives this information and starts the log analysis using optimal parameters.

The strong points of LogAnalysis as test application are that it is tunable using only application-dependent parameters, and that the repetitive nature of the log scansions performed allows its adaptation to system load without any run-time reconfiguration.

Framework performance evaluation: In order to evaluate the performance of the autonomic framework, we have performed a wide number of tests where multiple instances of the LogAnalysis client application are launched. As mentioned above, each of these instances performs a log scansion every T = 240 seconds. We assume that all the compute and network load is directly or indirectly generated by the LogAnalysis clients (i.e., the impact of further active sources of load is negligible), and that the log scansions generated by the active clients can be considered synchronized (i.e., they start at approximately the same time). Under these assumptions, a variation of system load can be emulated by varying the number of active LogAnalysis clients. For testing purposes, we will use the reference system load profile shown in Fig. 3 (solid line). At successive steps in time (the interval between two steps is equal to T) the number of active LogAnalysis clients varies according to the diagram, thus varying the system load. In order to get results that can be easily interpreted, we assume that the load prediction made by the Decision Unit is the one shown with the dotted curve in the same diagram. The predicted and actual loads are almost always very similar, with the exception of the time steps 12 and 13 (slight misprediction), and of the steps between 19 and 22, where a completely wrong prediction is purposely used to study its effect on performance. In any case, the only measured result will be the mean response time of all active clients, *for a single log scansion*.

The environment used to test the application is the Orion Cluster, at the Second University of Naples. The cluster, based on the Rocks Linux Distribution, is composed of a front-end (dual Intel Xeon 2.8 GHz) and a blade system with 7 compute nodes (dual Xeon 2.8 GHz). During the tests, we assign only one role to each compute node. Hence each node can *exclusively* be or a generic service provider (provider that offers the services used by the target application), or a MAWeS server (provider that offers the MAWeS services), or a client (on which the application client is executed). In the whole test environment there can be multiple servers (of every type) and clients.

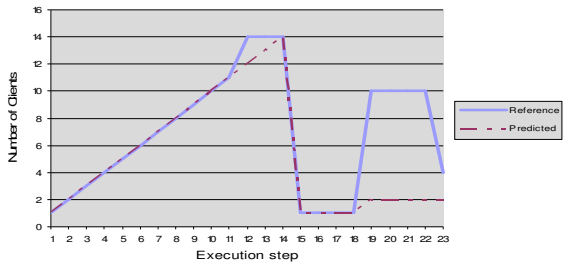

Fig. 3: Reference and predicted load profiles

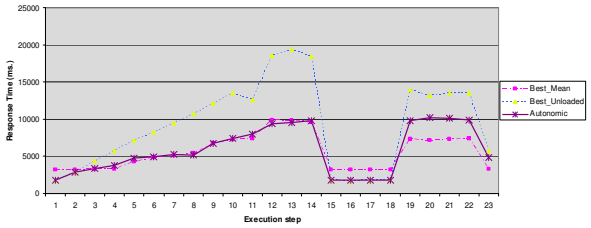

Fig. 4: Autonomic response time vs. two fixed configurations

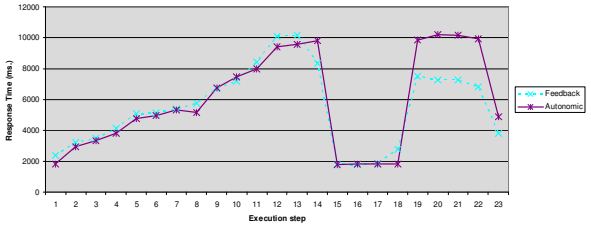

Fig. 5: LogAnalysis response time, feedforward vs. feedback

In Fig. 4, the LogAnalysis application autonomic response time (for a single log scansion) is compared to the behavior found for the same application in the absence of autonomic optimizations, i.e., using a fixed parameter configuration at all time steps and for all load conditions. These two "reference" configurations correspond to values of the optimization parameters `NumberOfProcesses`, `NumberOfServers` and `ServerChoicePolicy` (4, 4, *static*) and (1, 4, *dynamic*). These two configurations are denoted by *Best_Unloaded* and *Best_Mean*, as they correspond to the parameter set able to obtain the shortest response time in an completely unloaded environment, and to the one that leads to the shortest mean response time throughout all the tested load conditions, respectively. The diagram shows that whenever reasonably good load predictions are used for optimizations, the autonomic version of the application performs much better than *Best_Unloaded*, with response times very close to the static configuration that *a posteriori* appears to be the best on the average (*Best_Mean*). Stated another way, the overhead introduced by the framework is negligible. Even in the presence of a completely mispredicted load, the autonomic performance is acceptable.

As mentioned earlier in this study, a distinctive feature of our proposal is the use of a feedforward approach for autonomic optimizations. The framework exploits the simulation tools to find the best working condition for the system not in the current status, but in the one predicted for the future, anticipating the need. We have compared the results obtained using our feedforward method to the ones that could be obtained exploiting feedback data for optimizations. In the latter case, the Decision Unit makes its optimizations using the current system state, not the predicted one. Fig. 5 shows the LogAnalysis application response time using feedforward and feedback approaches, under the same load conditions and predictions (Fig. 3). The diagram shows that the feedback-based approach performs better than the feedforward only when the load is mispredicted.

RELATED WORK

Most of the times, autonomic computing models rely on some sort of feedback control mechanism to provide self-configuration and/or self-optimization capabilities[5]. Following up the seminal study that firstly introduced the use of predictive autonomicity[12], further work based on feedforward control (or on the joint use of feedforward and feedback) is currently being developed[9].

To authors' knowledge, the adoption of simulation-based tools in the context of autonomic computing, which is the main contribution of this study, has never dealt with before. However, it should be pointed out that the use of simulation for Web Services, and of Web Services interfaces to simulation, is described in reference 3. The necessity of the addition of autonomic behavior to web services and to the underlying messaging substrate for reliability purposes is instead dealt with in reference 16 and 18, respectively.

RESULTS AND CONCLUSION

In this study we propose an innovative approach to development of self-optimizing autonomic systems for Web Services architectures, based on the adoption of simulation for performance prediction and feedforward control. We have developed a set of web services that enable web applications to obtain performance predictions using the MetaPL/HeSSE methodology. The web service interface to the simulation environment has made possible the implementation of a framework for the development of autonomic self-optimizing applications. The framework exports a standard client application to final users, which enables the interfacing of web applications to the simulation environment. MetaPL Web Services and autonomic extensions enable the user to enrich an existing application with a model that can be simulated to obtain performance predictions.

The web application thus obtained runs, queries the framework for optimal parameter values, and then executes in an optimized way.

The proposed framework opens a new way for the development of autonomic systems that not only query the local environment for self-optimizing information, but also predict future load and network conditions, preparing it to well-tuned executions under in the future. We are currently involved in an expansion of the framework described here. The short-term objective is to add new, more complex, optimization rules for parameters. Successively, we will test the validity of the approach applying the optimizations also at service and server levels.

REFERENCES

1. Balasubramanian, V. and A. Bashian, 1998. Document management and web technologies: Alice marries the Mad Hatter. In Commun. ACM, 41: 107-115, ACM Press.
2. Booth, D., H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard, 2004. Web Services Architecture. W3C Web Services Architecture Working Group.
3. Chandrasekaran, S., J.A. Miller, G. Silver, I. Arpinar and A.P. Sheth, 2003. Performance analysis and simulation of composite web services. Electronic Markets, 13: 120-132, USA. Routledge.
4. Chandrasekaran, S., G. Silver, J.A. Miller, J. Cardoso and A.P. Sheth, 2002. Web service technologies and their synergy with simulation. Proc. Winter Simul. Conf., pp: 606-615, San Diego (CA), USA. ACM.
5. Diao, Y., J.L. Hellerstein, S. Parekh and J.P. Bigus, 2003. Managing web server performance with AutoTune agents. IBM Syst. J., 42: 136-149. IBM Corp.
6. Birman, K.P., R. van Renesse and W. Vogels, 2004. Adding high availability and autonomic behavior to web services. Proc. 26th Intl. Conf. on Softw. Eng., pp: 17-26, Edinburgh, UK. IEEE Computer Society.
7. IBM Corp., 2004. An architectural blueprint for autonomic computing. IBM Corp.
8. Kephart, J.O. and D.M. Chess, 2003. The vision of autonomic computing. Computer, 36: 41-50, IEEE Press.
9. Zhang, Y., A. Liu and W. Qu, 2004. Software architecture design of an autonomic system. Proc. 5th Australasian Workshop on Softw. and Syst. Arch., pp: 5-11, Melbourne, Australia.
10. Jacob, B., S. Basu, A. Tuli and P. Witten, 2004. A First Look at Solution Installation for Autonomic Computing. IBM Corp.
11. Jacob, B., R. Lanyon-Hogg, D.K. Nadgir and A.F. Yassin, 2004. A Pratical Guide to IBM Autonomic Computing Toolkit. IBM Corp.
12. Russell, L.W., S.P. Morgan and E.G. Chron, 2003. Clockwork: A new movement in autonomic systems. IBM Systems J., 42: 77-84, IBM Corp..
13. Mazzocca, N., M. Rak and U. Villano, 2001. MetaPL a notation system for parallel program description and performance analysis parallel computing technologies. LNCS, 2127: 80-93, Berlin (DE). Springer-Verlag.
14. Mazzocca, N., M. Rak and U. Villano, 2002. The MetaPL approach to the performance analysis of distributed software systems. Proc. of 3rd Intl. Workshop on Softw. and Perf., pp: 142-149, IEEE Press.
15. Mancini, E., N. Mazzocca, M. Rak, R. Torella and U. Villano, 2005. Performance-driven development of a web services application using MetaPL/HeSSE. Proc. 13th Euromicro Conf. on Par., Distr. and Network-based, pp: 12-19, Lugano, CH.
16. Mazzocca, N., M. Rak and U. Villano, 2000. The transition from a PVM program simulator to a heterogeneous system simulator: The HeSSE project. LNCS, 1908: 266-273, Berlin (DE). Springer-Verlag.
17. Mazzocca, N., M. Rak, R. Torella, E. Mancini and U. Villano, 2003. The HeSSE simulation environment. Proc. European Simulation and Modelling Conf., pp: 270-274, Naples, Italy.
18. Fox, G., Pallickara, S., Pierce, M. and D. Walker, 2003. Towards dependable grid and web services. Ubiquity, 4: 25, 3-13. ACM Press.