Abstract—The highly distributed nature and the load sensitivity of Service Oriented Architectures (SOA) make it very difficult to guarantee performance requirements under rapidly-changing load conditions. This paper deals with the development of service oriented autonomic systems that are capable to optimize themselves using a feedforward approach, by exploiting automatically generated performance predictions. The MAWeS (MetaPL/HeSSE Autonomic Web Services) framework allows the development of self-tuning applications that proactively optimize themselves by simulating the execution environment. After a discussion on the possible design choices for the development of autonomic web services applications, a soft real-time test application is presented and the performance results obtained in a composite-service execution scenario are commented.

I. INTRODUCTION

Nowadays Service Oriented Architectures (SOA) and their Web Services implementations are among the most relevant software solutions. But while the technology standards SOAP, WSDL, . . . ) and their implementations (Axis, .NET, . . . ) are mature enough today, the definition of good methodologies for development of efficient services and web-based applications is still an open issue. Management of heterogeneous, geographically distributed systems is a hard task, as the development of applications whose performance can be predictable is often impossible. In practice, the only solution to guarantee critical performance 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 aims to bring automated self-management capabilities into computing systems [1]–[4]. The name “autonomic” derives from the biological sciences, where it indicates the human autonomic nervous system. Autonomic capabilities are usually classified in four different categories: self-configuring, self-healing, self-optimizing and self-protecting. A self-configuring system can dynamically adapt to changing environments, a self-healing one can discover, diagnose and react to disruptions, a self-optimizing one can monitor and tune resources automatically and a self-protecting system can anticipate, detect, identify and protect against threats.

Even if currently there are many simple examples of application of autonomic concepts, and a toolkit for building autonomic systems is available from IBM, all known solutions are fairly “young”, and rather unstable [5]–[7]. 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 points out configuration or performance problems. The system automatically reacts applying new configurations or tuning policies. Recently a different approach has been proposed, which is based on predictive autonomicity, i.e., on feedforward control [7]. In this case, the system detects and forecasts cyclic variations in parameters and their impact on performance, and dynamically self-tunes, anticipating the need. However, application development tends to be harder, because the autonomic behavior needs to be designed in an integrated way with application and system.

In companion papers [8], and more thoroughly in [9], we have presented a framework (MAWeS) that may be integrated with the application. This framework makes it possible to predict application performance in the expected system status before its startup, and to optimize its execution in the future load conditions using both system- and application-dependent parameters. Our proposal is based on what we have called the MetaPL/HeSSE methodology [10]. This uses a meta-language (MetaPL) to describe the software and to identify the critical parameters that influence the performance (e.g., bandwidth, number of application threads, . . . ), and exploits the predictions produced by a simulation tool (HeSSE) to optimize the values. The software is thus proactively adapted to the forthcoming environment changes.

In the previously mentioned papers, [8] and [9], we have presented the MAWeS framework and a toy application where autonomicity is exploited at application level, supporting self-tuning applications that extend a given client. In this paper, we instead describe the development process of self-optimizing predictive autonomic Web Services, exploiting the MAWeS framework at service level, and exploring the more complex case of composite services.

The rest of the paper is organized as follows. Firstly we describe MetaPL, HeSSE, MAWeS and a review of related work. Then we introduce the framework methodology and its application at service level. Finally we present the experimental results, and our future work and the conclusions are outlined.
II. MAWeS METHODOLOGY

The MAWeS framework has been developed to support the predictive autonomicity in Web Service based architectures. It is based on two existing technologies: the MetaPL language [11] and the HeSSE simulation environment [12]. The first is used to describe the software system and the interactions inside it; the second, to describe the system behaviors and to predict performances using simulation.

HeSSE is a simulation tool that allows to simulate the performance behavior of a wide range of distributed systems for a given application, under different computing and network load conditions. It makes it possible to describe Distributed Heterogeneous Systems by interconnecting simple components, which reproduce the performance behavior of a section of the complete system (for instance a CPU, a network . . . ).

MetaPL is an XML-based meta-language for parallel programs description, which, like other prototype-based languages, can be used when applications are in design phase or they are not completely available [10], [11]. It is language-independent, and can be adapted to support different programming paradigms; it is structured in layers, with a core that can be extended through Language Extensions, implemented as XML DTDs. These extensions 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.

Filters can be used to generate views for program comprehension or documentation purposes, but also to produce traces of program activity annotated with timing information. These traces are amenable to direct simulation in HeSSE, thus making it possible to obtain fairly reliable predictions of program performance, even at the very first steps of code development [13].

A. MAWeS

Through the execution of multiple simulations, corresponding to different parameterized configurations, it can choose the parameter set that optimizes the software execution according one or more criteria (e.g., shortest execution time).

Depending on whether the optimization choices are performed at the launch of an application or at run-time, the necessity to simulate a high number of system configurations may lead to increased execution latencies, or to loss of reactivity of the autonomous system. In order to reduce the simulation overhead to a minimum, a large set of information is collected off-line (i.e., during the set-up of the application). This makes possible to reduce the number of simulations to be performed at run-time. In any case, the developer can help MAWeS to reduce the number of candidate configurations and thus simulation overheads, indicating explicitly in the MetaPL code the upper and lower bound for system parameters. For example, the developer can supply maximum and minimum number of computing nodes to be used for executing the autonomic application, letting the autonomic framework decide by examining HeSSE performance predictions if in the coming CPU and network conditions it is better to increase parallelism or to reduce the amount of communication.

MAWeS is structured in three layers (Fig. 1). The first one is the front-end, which contains the software modules used by final users to access the MAWeS services. The second one is the core, which includes the components that manage MetaPL files and make optimization decisions. The last one contains the Web Services used to obtain simulations and predictions through MetaPL and HeSSE.

The MAWeS frontend includes a standard client application interface, MAWeSclient, providing the general services that can be used and extended to develop new applications. 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 MAWeSclient. The MetaPL/HeSSE WS interface defines a set of services that make it possible to automatize the generation of performance predictions by MetaPL/HeSSE. It is worth pointing out that all simulations and optimization steps are made on the hardware/software environment indicated by the developer through the MetaPL description. Possible successive changes in the execution environment involve corresponding modifications of the MetaPL code.

The activity diagram of Fig. 2 shows the actions of an application exploiting the MAWeSclient interface, which asks the framework for configuration optimizations and reconfigures itself.

The MetaPL description must suitably identify the set of parameters that can be modified by the optimization engine. MAWeS will automatically perform a set of simulations varying the values of these parameters to find the optimal set of values. The user can specify the tunable parameters by means of the autonomic MetaPL extension, which defines

![Fig. 1. The three layers of the MAWeS Framework. The client functionalities in the Frontend can be integrated in applicative software](image-url)
new MetaPL elements for the Mapping section [8], [9]. The extension introduces in the meta-language the Autonomic tag, which describes the target simulation configurations that can be used for application execution (Fig. 3). Another tag, named Parameter, identifies a variable that affects the overall performance, whose optimal value has to be evaluated by the framework. The parameter domain can be specified in terms of range (minimum and maximum), or in terms of a list of valid values. The tag Constrain allows to introduce limits in the parameters domain using mathematical relations between the parameters. As mentioned before, the use of narrow variation ranges may help to prevent high autonomic overheads. Finally, the tag Target identifies the simulation output parameter that is the optimization target (e.g., the response time to be minimized).

Fig. 3. Fragment of a MetaPL description of a Web Services client with autonomic extensions. The parameters definition (processes and band) and the constraints are shown in italics

B. Related work

The majority of autonomic computing models rely on feedback control mechanism to provide self-configuration and/or self-optimization capabilities (see for example [14]). The use of predictive autonomicity was firstly introduced in [7]. Further research is currently performed on the use of feedforward control, and on the joint use of feedforward and feedback [4]. The adoption of simulation-based tools in the context of autonomic computing has been introduced by the authors in [8] and [9]. Previously, the use of simulation for Web Services, and of Web Services interfaces to simulation, has been described in [15].

The use of composite services, i.e., of services exploiting several sub-services or sharing the task among them, has been studied mainly in the context of Workflow Management. Standard languages such as BPELWS, WSFL o XLANG have been developed to support service composition through choreography and orchestration techniques. [16]. The difficulties to obtain satisfactory performance in such systems make them ideal candidates to MAWeS autonomic optimizations.

III. AUTONOMIC WEB SERVICES

Web Applications are composed of two main elements: the Client, the software module that interacts directly with the final user, and the Services, a set of distributed functions and objects with Web Service interface. Usually, but not mandatorily, the client module maintains the application status during the execution and exploits the services for elaboration tasks (i.e., manipulation of data, access to data bases, ...). The development of a new application consists of the definition of new clients, which interact with the final user, of a new set of services, as reusable as possible, and of the aggregation of the existing ones.

Web Services systems are affected by significant call stack overheads, and are difficult to tune when the dimension of the hardware/software system grows, especially in the case of architectures based on composite services. In order to achieve good performance from a web application, 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 the set of offered services;
- **Application Level**: the autonomic system affects user applications, modifying their resource use and the order of the actions they perform.

While the Server level self-tuning activities are usually managed by system administrators, both Service and Application level optimizations can be adopted by web application developers. Our previous papers [8], [9] were focused on the use of MAWeS (MetaPL/HeSSE Autonomic Web Services) to provide autonomicity at Application Level. In this case, the application developer exploits the predefined client provided by the framework. The client invokes the optimization services, and then the actual services with optimal parameters: the control of the application performance is up to the client, as shown in Fig. 4(a).

![Fig. 4. The application-level and the Service-level approach in MAWeS.](image)

**Fig. 4.** The application-level and the Service-level approach in MAWeS. In (a), the web application client contains the logic needed to control the optimizations. In (b), each service chooses when to optimize its parameters when to start the system performance analysis. We considered the following strategies for triggering the optimization:

- **Deploy**: optimization takes place only at service deployment;
- **Service Call**: each time a client calls the service, it self-optimizes, tuning itself to the current system status;
- **Reactive**: optimization takes place in correspondence with particular events, such as a timer expiration or a status notification from a load monitor.

The first strategy, **Deploy**, makes the tool unable to react to status changes in the environment: once the service is deployed, it is impossible to re-tune it. Hence it is suitable only for very stable systems, where the status conditions do no change sensibly during the whole application execution cycle.

The second strategy, **Service Call**, imposes high overhead to the service execution: every time that the service is started, a new set of simulations are to be performed. It can be useful in the case of long-running services, where the overhead is dwarfed by computation times, or more in general when one may aspect from optimizations performance advantages that are higher than the introduced overhead.

The most flexible solution is using the **Reactive** approach. In this case, optimizations choices can be performed only when they are really necessary, as the consequence of “significant” system status changes. The optimization trigger can be a notification event chosen by the application developer. This notification can be generated, for example, using a *workload profile*, that makes it possible to predict the times at which the system workload *should* change.

The autonomic Web Services architecture that will be considered in the rest of this paper is the server-side of Fig. 4(b), with the MAWeS client integrated into the web service (Fig. 5). When, according the optimization schedule that derives from one of the three strategies mentioned above, the MAWeS client starts the optimization services, the actions a-g in the figure are performed, and a “fresh” set of optimal parameters is produced. On the other hand, calls from the final user’s web client trigger the actions 1-4 in the figure, which execute the end-user service using the latest known set of optimal parameters.

**IV. TEST APPLICATION: WORKHOUR**

In order to show by means of a practical example how the autonomic capabilities provided by MAWeS can be integrated in service-oriented architectures, we have developed a synthetic test application, WorkHour.

Many applications, especially in the entertainment area (video decoders, network gaming systems, ...), can tolerate a momentary loss of output quality, in cases of brief computing resource overloads. For instance, an “avatar” that moves in a video game world can choose a non optimal path to find its objective without appreciable variations of the user experience, or a video stream system can send or decode some frames at lower quality, but nevertheless guarantee its services. Often overload phenomena are not completely random, but they follow some kind of empirical law. For instance, the web or
streaming servers have an increase of the number of accesses every day at the same hour.

WorkHour is a synthetic test application representative of such a soft real-time system behavior. It simulates the execution of a set of work units which should terminate within a predefined time slot, otherwise a (tolerable) output degradation is observed. Basically, WorkHour exposes a service that performs the conversion of set of images to a compressed jpeg format using the standard Java image libraries. The faster the conversion is executed, the better the output quality. In practice, the optimization criterion to be used is the minimization of execution time, or (what is the same) the maximization of the converted frame rate. In all the tests presented below, all frames have the same size (720x575 pixels). When the service is called, the back-end retrieves these images from an external repository, and, after the computation has been performed, stores the result frames on the same repository. WorkHour entails network load to read and to store the data from/to the repository (and for subservice calls, in the case of the composite services architecture), plus CPU load to perform the conversion. As far as simulation is concerned, we make the simplifying assumptions that the amount of data transmitted on the network and the CPU time required for conversion are constant for every image.

We have implemented ad hoc components (Work Servers) that execute these units. The application has been designed using a web service architecture and the methodology described in previous sections, with the structure shown in Fig. 5.

In order to experiment with both monolithic and composite-services architectures, the WorkHour end-user service exposes one web interface with two services:

- Convert, which performs the computation atomically (i.e., without resorting to subservices);
- ConvertCS, which is not able to perform the computation on its own, but distributes the frames to be processed to one or more instances of Convert.

Depending on whether the client interface of WorkHour calls ConvertCS, which distributes the work to subservices, or directly Convert, which computes with no use of subservices, the conversion performed by WorkHour will be obtained by a simple or a composite-service architecture. The tests shown in the following have been carried out on the composite-service version, as the atomic version has been only exploited for debugging for performance comparisons. Figures 6 and 7 show two fragments of the MetaPL description of WorkHour, containing the web interface and the autonomic section, respectively.

```
<MetaPL>
  <WebService name="WorkHour">
    <wsMethod name="ConvertCS">
      <WsParam>numframes</WsParam>
      <Block>
        <Loop iteration="frameblock" max="numframes/subservices">
          <WsCall ws="WorkHour" host="host" method="Convert" param="frameblock">
            <param>frameblock</param>
          </WsCall>
        </Loop>
      </Block>
    </wsMethod>
    <wsMethod name="Convert">
      <WsParam name="frameblock" />
      <Block>
        <Receive load="numframes*framesize" method="load" />
        <CodeBlock coderegion="conversion" time="numframes*10" />
        <Send dim="numframes*compressedsize" method="store" />
      </Block>
    </wsMethod>
  </WebService>
```

Fig. 6. Fragment of the MetaPL description of WorkHour: the web interface
In this Section the WorkHour application will be exploited to perform some experimentations. In the scenario we present, the hardware is dedicated to the use of WorkHour, which is the only source of CPU and network load, and the Service-call trigger strategy is adopted.

The tests that follow have been performed on a Rocks 4.1 cluster [17], composed of biprocessor nodes with XEON 2.8 GHz CPU, and 1 GB of RAM, and linked by Gigabit Ethernet. In all of the three scenarios considered, a dedicated node has been used to execute the MAWeS framework and to carry out simulations.

The scenario designed to analyze the behavior of composite services is the following. The system is dedicated, in that WorkHour is the only source of CPU and network load. In this conditions, probably the best strategy for triggering the autonomic optimizations is the one on Service Call. Due to the absence of external load, there cannot be unexpected and sharp load variations, and the autonomic system can manage to keep the load evenly shared by distributing suitably incoming calls to subservices. As mentioned later, the only drawback of this approach may be the increase of service start latency due to autonomic overheads. For this reason, this strategy is advisable for services with a relatively long service time, which can dwarf autonomic overheads.

With optimization on Service Call, before the actual service is started, MAWeS is queried for optimized parameters (in practice, the nodes where the request has to be serviced). The "reconfiguration" (which is not an actual reconfiguration, in this case) just manages the optimal configuration received from the autonomic framework, and the service is distributed accordingly to up to \( n \) subservices (Fig. 8).

The test shown here studies the evolution resulting from two service calls partially overlapped in time (Fig. 9). With optimization on service call, the optimization is made once for all at service start. The addition of further load to nodes being used for servicing would entail a loss of the optimized working conditions. Hence the system tries to service the second call using unloaded nodes (in this very simple case, only the third node in the figure). This optimization strategy is very simple and has the advantage to introduce autonomic overheads only at well-known moments in time, i.e., at the start of each service. However, it entails a growth of the service latency (time to actually start the service). In order to point out this overhead, and to evaluate its detrimental effect on service times, Fig. 10 shows the Gantt of the whole system activity (clients, services, MaWeS components) for the above described test.

As shown in the figure, as MAWeS evaluates beforehand possible CPU/network load configurations, the simulator is invoked only twice for each service call to simulate alternative CPU and network load configurations. This requires (using the current prototype implementation of the framework) a few seconds of activity. This is overhead that adds up to actual service times. For long-running services (as in the case presented in the Gantt, where the first service runs for over 150 s), this is acceptable. In other cases, it is better to switch to reactive triggering strategy for optimization.

VI. FUTURE WORK

Even if a prototype version of MAWeS is currently available, and makes it possible to evaluate the ideas on which the autonomic framework is based by performing experimentations, we are indeed convinced that our work on proactive autonomic optimization is at the very start. In particular, extensive testing remains to be done to assess the validity and the optimality of optimizations carried out in systems that are more large and widely distributed than the ones considered so far. At least in theory, scalability and stability of the system might be an issue, but we have not yet detected the

![Fig. 7. Fragment of the MetaPL description of WorkHour: the autonomic section](image-url)
problem in our tests. Currently, we are still involved in large-scale experiments, and we are working to the completion of a complete graphical development environment based on Eclipse [18].

As regards further fields of application for MAWeS, our interest is principally focused on applications based on mobile agents. The mobile agents programming paradigm is an emerging approach for distributed programming, especially in the GRID and SOA contexts. In mobile agents systems, optimization problems are of great interest, due to the continuous changes of the execution contexts. An agent is able to suspend its own execution, to transfer itself to another agent-enabled host and to resume its execution at that destination. As a result, the platform usually does not have complete control over the execution node state. 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, and this is the point where MAWeS comes into play. This will be one of our future research fields.

VII. CONCLUSIONS

In this paper, we have described the process of building service-oriented autonomic applications that are able to self-tune by the MAWeS framework. After an overview of the methodology on which MAWeS is based, we have considered the possible design choices for the development of autonomic web services applications.

In the second part of the paper, we have instead presented a soft real-time synthetic test application, and shown some of the tests performed on the field, illustrating the obtained results. These results, though obtained on small-scale systems and without considering exhaustively all possible scenarios, point out that the approach is valid, and deserving further investigation.

REFERENCES
