An Autonomic Computing Architecture for Self-* Web Services

Walid Chainbi[1], Haithem Mezni[2], and Khaled Ghedira[3]

[1] Sousse National School of Engineers/LI3 Sousse, Tunisia
Walid.Chainbi@gmail.com
[2] Jendouba University Campus/LI3 Jendouba, Tunisia
haithem.mezni@fsjegj.rnu.tn
[3] Institut Supérieur de Gestion de Tunis/LI3 Tunis, Tunisia
Khaled.Ghedira@isg.rnu.tn

Abstract. Adaptation in Web services has gained a significant attention and becomes a key feature of Web services. Indeed, in a dynamic environment such as the Web, it's imperative to design an effective system which can continuously adapt itself to the changes (service failure, changing of QoS offering, etc.). However, current Web service standards and technologies don't provide a suitable architecture in which all aspects of self-adaptability can be designed. Moreover, Web Services lack ability to adapt to the changing environment without human intervention. In this paper, we propose an autonomic computing approach for Web services' self-adaptation. More precisely, Web services are considered as autonomic systems, that is, systems that have self-* properties. An agent-based approach is also proposed to deal with the achievement of Web services self-adaptation.

Keywords: Web service, autonomic computing systems, self*-properties.

1 Introduction

With the rapid growth of communication and information technologies, adaptation has gained a significant attention as it becomes a key feature of Web services allowing them to operate and evolve in highly dynamic environments. A flexible and adaptive Web service should be able to adequately react to various changes in these environments to satisfy the new requirements and demands.

When executing Web services, network configurations and QoS offerings may change, new service providers and business relationships may emerge and existing ones may be modified or terminated. The challenge, therefore, is to design robust and responsive systems that address these changes effectively while continually trying to optimize the operations of a service provider.

So, while we obviously need effective methodologies to adapt web services to a dynamically changing environment, we also need elegant principles that would give web services the ability to continue seeking opportunities to improve their behavior and to meet user needs. To meet these goals, we propose an autonomic computing architecture for self-adaptive web services. More precisely, we consider Web services as autonomic computing systems.

A.V. Vasilakos et al. (Eds.): AUTONOMICS 2009, LNICST 23, pp. 252–267, 2010.

Autonomic computing is to design and build computing systems that can manage themselves [1]. These systems are sets of called autonomic elements whose interaction produces the self-management capabilities. Such capabilities include: self-configuration, self-healing, self-optimization, and self-protection [2].

- Self-configuration by adapting automatically to the dynamically changing environment.
- Self-optimization by continually seeking opportunities to improve performance and efficiency.
- Self-healing by discovering, diagnosing and reacting to disruptions such as repairing service failure.
- Self-protection by defending against malicious attacks or cascading failures.

Other instantiations of self-managing mechanisms have been also adopted namely autonomy of maintenance by Chainbi [3], and system adaptation and complexity hiding by Tianfield and Unland [4].

The rest of this paper is organized as follows: section 2 gives a background material on autonomic computing and shows how autonomic computing capabilities may be applied in Web services. In section 3, we describe our solution for self-* Web services. Section 4 deals with a case study. In section 5, we compare the proposed study to related work. Section 6 presents some hints justifying the possible implementation of the presented autonomic architecture via an agent-based approach. Section 6 compares the proposed study to related work. The last section presents the conclusion and the future work.

2 Autonomic Computing and Web Services

2.1 Autonomic Computing

Autonomic Computing is started by IBM in 2001 and is inspired by the human body's autonomic nervous system [2]. It is a solution which proposes to reallocate many of the management responsibilities from administrators to the system itself.

Autonomic computing deals with the design and the construction of computing systems that posses inherent self-managing capabilities. Such systems are then considered as autonomic computing systems. Each autonomic system is a collection of autonomic elements – individual systems constituents that contain resources and deliver services to humans and other autonomic elements. It involves two parts: a managed system, and an autonomic manager. A managed system is what the autonomic manager is controlling. An autonomic manager is a component that endows the basic system with self-managing mechanisms such as self-configuration, self-optimization, self-healing, and self-protection.

The autonomic computing architecture starts from the premise that implementing self-managing attributes involves an intelligent control loop [2] (see figure 1). This loop enables the system to be self-*, and is dissected into three parts that share knowledge:

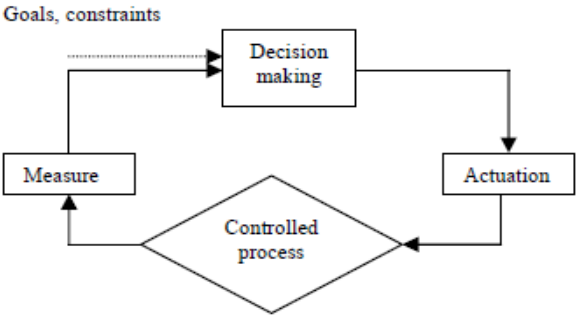

Fig. 1. A control loop (adapted from [4])

- The measure part provides the mechanisms that collect, aggregate, filter, and report details (e.g., metrics) collected from the controlled process.
- The decision part provides the mechanisms to produce the action needed to achieve goals and at the same time respect the constraints.
- The actuation part provides the mechanisms that control the execution of actions.

With this characterization in place, the integration of autonomic computing may be envisioned by two ways, namely (a) embedding the autonomic manager into the Web service, and (b) using autonomic managers as an external layer which provides the autonomic behavior for the Web service. The latter approach treats the Web service as a "black box" surrounded by autonomic managers controlling the state of the Web services and performing actions each of which can configure, optimize, heal, or correct the Web service. This approach requires the development of specific interfaces facilitating the interaction between the autonomic manager part and the managed part of the Web service. In this paper, we adopt the latter approach because it seems to be more appropriate since it separates the monitoring problem from the application specification. Accordingly, it ensures separation of concerns in the development process. Hence, the proposed architecture clearly separates the business logic of a Web service from its adaptation functionality.

2.2 Self-* Properties in Web Services

We define self-adaptive Web services as Web services supporting the autonomic computing properties which are often referred to as self-* properties, and are the followings.

- ***Self-configuration*:** a Web service may be a set of interacting Web services which in turn may interact with external applications with different interfaces and protocols. A new component Web service has to incorporate itself seamlessly and the rest of the system will adapt itself to its presence. For example, when a new Web service is introduced into a composite Web service, it will automatically learn about and take into account the composition and the configuration of the Web service, so that it can process services mismatches between interfaces and protocols by taking mediation actions.

- ***Self-healing*:** In case of problems such as service failure or QoS violation, a Web service has to perform recovery actions including retry execution, substitute candidate service, etc.
- ***Self-optimization*:** Web services have to continually seek opportunities to improve their own performance and efficiency. For example, a component Web service may be substituted with another one guaranteeing a better QoS and taking into account runtime execution context and other constraints.
- ***Self-protection*:** Web services can interact with other services or Web applications. They have to identify and detect intrusions and defend themselves against attacks (e.g., message alteration or disclosure, availability attacks, etc.) by defining and integrating some security policies.

It is important to note that these capabilities may be heavily interrelated with one another in the Web service adaptation. For example, consider a system that fails to invocate a component Web service. The adaptation process starts by detecting and diagnosing the failure. Then the recovery action is to substitute this component Web service (i.e., self-healing action). The system selects the suitable service based on current state of the environment (i.e., self-optimization action). If the substituted and the substituting services interfaces don't match, some mediation actions have to be taken (i.e., self-configuration action).

3 Autonomic Web Service Architecture

Dealing with self-* in a Web service means simply that the Web service act without the direct intervention of any other external agent (including but not limited to, a human) in order to meet self-* properties. Accordingly, autonomy is required for Web services to self-* themselves to the different events occurring in their environment. Autonomy is also necessary for an autonomic computing system. Indeed, system self-* has to be carried out without requiring the user's consciousness [4].

With this characterization in place, the match process in favor of an autonomic computing system solution is straightforward. A Web service is the system to be managed, and an autonomic manager is required to endow the Web-service with self-* capabilities such as self-configuration, self-healing, self-optimization, and self-protection. Figure 2 represents an autonomic Web service architecture.

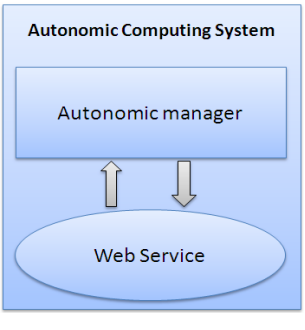

Fig. 2. An autonomic Web service architecture

An autonomic Web services environment may combine a variety of managed resources including autonomic Web services, processes, etc. These resources have different requirements and architectures which need autonomic managers with different capabilities allowing Web services to be self-*.

In case of executing a composite Web service, the management tasks of the component services are shared between a set of autonomic managers. The topology of the autonomic system is specified as a correlation between autonomic managers which perform many tasks such as managing the executing Web service, interacting with registries to select suitable web services or coordinate the work of other autonomic managers. The autonomic managers' work is orchestrated by a special autonomic manager which is responsible for the coordination of the basic Web services autonomic managers. Note that the autonomic manager coordinating the set of services' autonomic managers can be considered as a managed resource in case the associated composite Web service is used as a component service in another process. Figure 3 shows the structure of such a system.

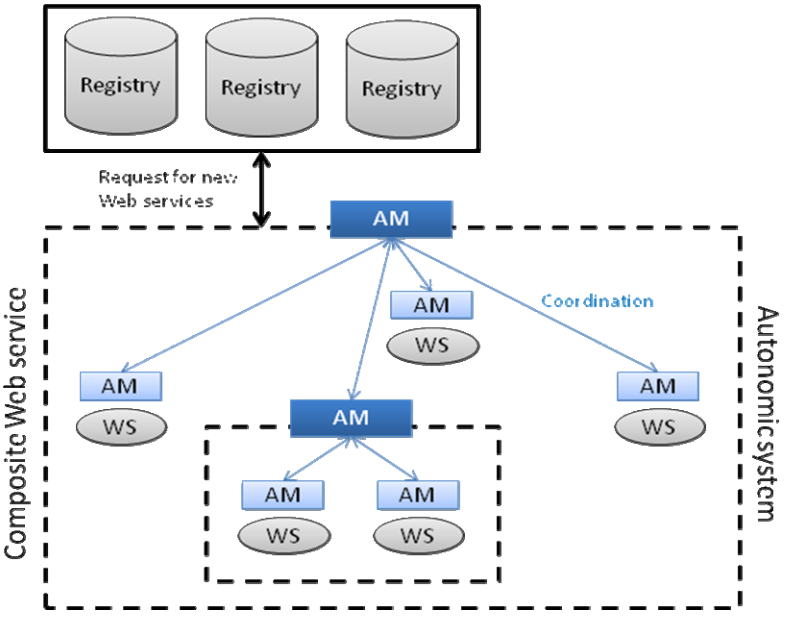

Fig. 3. An autonomic Web service system architecture

An autonomic manager has to interact with its external environment to be able to manage Web services efficiently. External environment includes registries and other autonomic managers. Interaction with registries enable autonomic managers to send a substitution request of a Web service, to select best available Web services for a new composition, to get new Web services opportunities, etc. Autonomic managers may also interact with registries (independently of the managed Web services) when detecting any change in the state of the environment, such as emergence of new nodes, termination of existing ones, etc. Such interaction allows autonomic managers to be

aware of the available resources for the adaptation process of the executing Web service or for a future management task.

Autonomic managers may also interact between each others to send information or to perform adaptation. For example, if a Web service's autonomic manager fails to adapt its managed Web service, it may send a request to its coordinating autonomic manager to execute adaptation actions at the composite service level (i.e., adaptation of the whole Web service). The same information's flow may occur between two coordinating autonomic managers in case the managed composite service is composed of some complex Web services.

4 Example of an Adaptation Scenario

The structure of the autonomic Web service system may change in run time to satisfy a self*-property. Figure 4 shows an example of adaptation scenario where a basic Web service is substituted by a composite Web service.

Using the search for music scenario, we show how an executing Web service may be adapted to meet the user needs and we show how the autonomic system is able to adjust its specification according to the changing conditions. In this scenario, the client wishes to listen to music and to download songs in the *rm* format while reading lyrics and information about the artist. Figure 5 shows the sequence diagram related to the adaptation scenario.

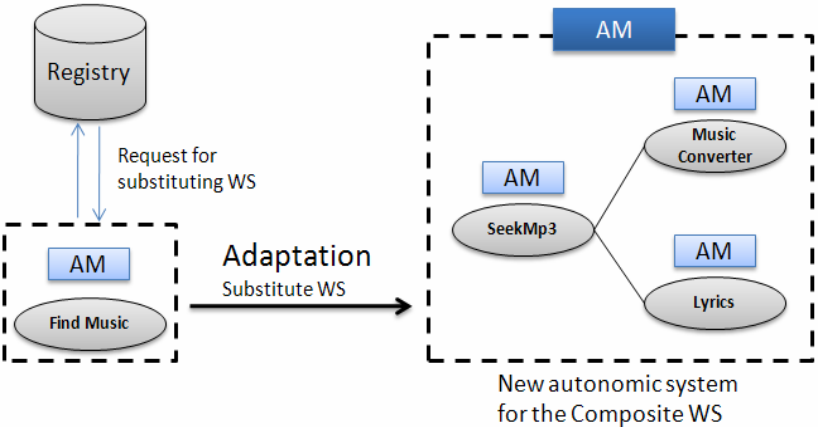

Fig. 4. Example of adaptation scenario

Consider the *FindMusic* Web service invoked by a user to look for a song. The service takes as inputs the song's title or the performer's name. Then, it shows the results according to the user's request. Since the user searches songs with a particular format, he may specify this format in the song's parameters. After selecting the desired song, the Web service proposes to play or to download the song. While listening, users have the possibility to read lyrics and artist information.

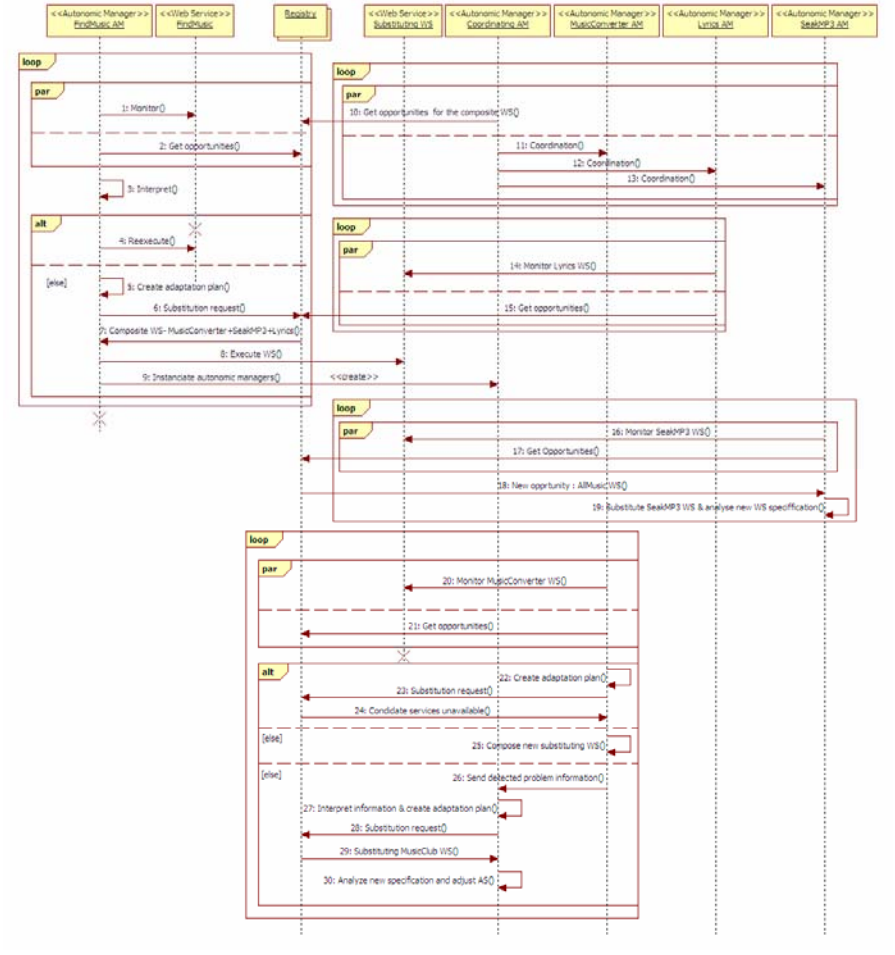

Fig. 5. Sequence diagram of the adaptation scenario

When starting the execution, the *FindMusic* Web service is associated to an autonomic manager, which performs the monitoring task and interaction with the external environment such as requesting registries to look for new opportunities or to execute adaptation actions.

The adaptation process starts when the autonomic system detects an event triggering a self-* action. Such an event may be an invocation failure of the *FindMusic* Web service. Therefore, the autonomic Web service system takes some recovery actions such as retrying execution. If the execution fails again then the autonomic system triggers a recovery action in order to replace this component with another one such as a composite Web service. The autonomic system interacts with Web services registries to perform the selection of the suitable service according to objectives of the failed *FindMusic* service.

Let the selected service be a composite web service, where the components are the *SeekMp3* Web service, the *MusicConverter* Web service and the *Lyrics* Web service. These Web services interact to satisfy user requests. The *SeekMp3* service receives the song title or artist name and returns a result which is a set of songs with different formats. Then *MusicConverter* service, based on the preferred format, is invoked to convert input files (results returned by the *SeekMp3* service) to the desired format. The *Lyrics* service uses the song's title to return lyrics of the song and information about the corresponding artist. Finally, outputs of the *MusicConverter* Web service and the *Lyrics* Web service (songs in *rm* format, lyrics and artist's information) are returned to the user.

The selected composite Web service is made up with three autonomic managers for managing the basic services and an autonomic manager for coordinating services' autonomic managers. The autonomic manager, responsible for monitoring the failed *FindMusic* service, is replaced by these autonomic managers that will manage the substituting composite service. Analysis and design of autonomic Web services is out of the scope of this paper which main content deal with an autonomic architecture to self-* Web services. Some hints are given in section 6.

Note that the desired service (that will substitute the failed *FindMusic* service) may be unavailable or may not exist. The autonomic system, then, has to interact with the registries to look for possible actions such as composing the substituting service. In case of replacing a composite web service by a basic one, the set of autonomic managers associated to the composite service are replaced by a single autonomic manager to manage the substituting basic service.

From a functional perspective, the substituting service meets the user needs and offers the same functionality of the failed *FindMusic* service. Rarely does service' WSDL interfaces match exactly. In our scenario, *SeakMp3* and *MusicConverter* services interfaces don't match. This requires taking some mediation actions to translate between the two service-interface signatures, so that interoperability can be made effective. For this, associated autonomic managers, based on their self-configuration capabilities, should interact to generate an adapter (e.g. a service) that mediates the interactions among the two *SeakMp3* and *MusicConverter* services.

Once the autonomic system is established, it continuously monitors the executing Web service to detect problems while trying to improve its performance. In our work, Web services self-optimization behavior is a combination of monitoring, selection and substitution capabilities. Self-optimization may occur in case of emergence of a new Web service with the same functionality and with a better quality. Regarding our executing Web service, possible optimization actions are: substituting one of the component services (*SeakMp3*, *MusicConverter* or *Lyrics*) or substituting the whole executing composite Web service.

Each autonomic manager must continuously try to improve the whole executing Web service performance by interacting with registries to get services opportunities. Indeed, each Web service's autonomic manager receives opportunities from registries and decides about substituting its associated Web service. In the same way, the coordinating autonomic manager should also interact with registries to look for a better Web service that may replace the whole executing web service.

Suppose that a change in the execution environment occurs (emergence of new Web services similar to the *SeakMp3* service). The autonomic manager associated to

the *SeakMp3* service receives the ranked candidate services list from registries and decide to replace the *SeakMp3* service with the basic *AllMusic* service. Then, it analyses the new specification of the executing composite Web service to look for any change. Since the *SeakMp3* Web service is replaced with a basic one, the autonomic manager decides that no changes have occurred in the service specification and keeps the current autonomic system specification.

Consider now, that the *MusicConverter* service is no longer available. The associated autonomic manager handles adaptation accordingly. For this, it implements a set of recovery actions that allows substituting the *MusicConverter* service. If no candidate service is available for substituting, the autonomic manager tries to apply another appropriate recovery action to let the execution successfully terminate. To that end, the autonomic manager features several recovery actions. Possible solutions are (a) composing a new Web service with the same functionality of the failed *MusicConverter* service or (b) replacing the whole executing composite service. In case of adopting the first solution, the autonomic manager uses its automatic service discovery and composition capabilities to interact with registries and perform the necessary repair actions. It may also choose the second solution and looks for assistance from its coordinating autonomic manager. This is by informing the coordinating autonomic manager about the detected problem (*MusicConverter* unavailability) and about the new information collected after trying to execute repair actions (unavailability of substituting services). So, based on information sent by the *MusicConverter* autonomic manager, and after interacting with registries, the self-healing behavior of the coordinating autonomic manager is to substitute the whole executing composite Web service with another one having the same goals. For this, the coordinating autonomic manager contacts the registries to get a set of candidate services similar to the desired one (the whole Web service) and selects the best available service: the *MusicClub* Web service. Once the *MusicClub* service starts to execute, the autonomic system adjust itself according to the new specification of the executing Web service by instantiating new autonomic managers or deactivating existing ones. In case of replacing the composite Web service by a basic one, the set of autonomic managers associated to the composite service are replaced by a single autonomic manager to manage the substituting basic service.

5 Related Work

The main purposes of service adaptation vary from ensuring interoperability to service recovery and optimization and context management. Some approaches address the problem of interoperability due to interfaces and protocols heterogeneity. Recovery deals with techniques for detecting problems in services interaction and searching alternative solutions. Optimization is about discovering and selecting the suitable Web service with respect to QoS offerings and user needs. Finally, solutions for context change aim to optimize the service function of their execution context. Here are some works on service adaptation:

The WS-Diamond Project [5] aims at the development of a framework for self-healing Web services, that is, services able to self monitor, to self-diagnose the causes of a failure, and to self-recover from functional and non-functional failures. In [6], the

solution for self-healing of BPEL processes is based on Dynamo, a monitoring framework, together with an AOP extension to ActiveBPEL, and a monitoring and recovery subsystem that uses Drools ECA rules.

In [7], a methodology and a tool for learning the repair strategies of WS to automatically select repair actions are proposed. The methodology is able to incrementally learn its knowledge of repairs, as faults are repaired. Thus, it is at runtime possible to achieve adaptability according to the current fault features and to the history of the previously performed repair actions. In [8], the authors propose a methodology for the automated generation of adaptors capable of solving behavioral mismatches between BPEL processes. [9] introduces PAWS, a framework for flexible and adaptive execution of managed WS-based business processes. In the framework, several modules for service adaptation (mediation engine, optimization and self-healing) are integrated in a coherent way.

In [10], the authors developed a staged approach for adaptive Web service composition and execution (A-WSCE) that cleanly separates the functional and non-functional requirements of a new service, and enables different environmental changes to be absorbed at different stages of composition and execution.

In [11], the authors are focusing on run-time adaptation of non-functional features of a composite Web service by modifying the non-functional features of its component. The aspect oriented programming technology is used for specifying and relating non-functional properties of the Web services as aspects at both levels of component and composite services.

While current approaches address significant subsets of adaptation requirements, they have some drawbacks including the degree of automation, few techniques for capturing non-functional properties, etc. The autonomic approach presented in this paper deals with the different facets of adaptation since its purpose is the design and the construction of Web services that posses inherent self-* capabilities.

Furthermore, there is no existing approach addressing the adaptation cross all the functional layers of the service based systems (i.e., the business process layer, the service composition layer, and the service infrastructure layer) since all the approaches address only a particular functional layer. For example, [10] and [11] deal with the infrastructural layer whether the composition layer was dealt with in [8] and [9]. In addition, existing approaches try to integrate a maximum of requirements in order to have a complete framework. For this purpose, our main concern is to propose a general autonomic architecture that provides self-* capabilities and meets the most important adaptation requirements without affecting services consistency and by preserving the robustness of the applications. Moreover, none of the existing approaches have studied the complexity in the implementation, that is, hiding the complexity from user and how much the adaptation is complex at any time of the application lifetime. Autonomic computing provides self-adaptation while keeping its complexity hidden from the user [4].

In our work, considering Web services as autonomic systems, offers many advantages. First, unlike existing approaches, the management task is shared between a set of autonomic managers, each of them is associated to a Web service. This leads to an effective monitoring and consequently to a high degree of adaptation.

6 Implementation Issues

In this section, we deal with the technical machinery to achieve the self-adaptation. We adopt an agent-based solution for the autonomic Web-service system. The integration of agent-based computing into the framework of autonomic computing may be envisioned by two ways, namely (a) integrating the autonomic cycle into the system, thus in a certain sense embedding the autonomic manager into the managed system and adopting an agent solution for the whole, and (b) using agents as an external layer which provides the autonomic behavior. We propose to adopt an agent solution for the whole namely the managed part of the Web service and the manager part. Consequently, the interaction between the managed and the managing parts of the system become easier. This is mainly due to the homogeneity of the adopted solution (an agent is the unit of design).

In any design process, finding the right models for viewing the problem is a main concern. In general, there will be multiple candidates and the difficult task is picking the most appropriate one. Next, we analyze the high degree of match between the characteristics of agent systems and those of autonomic systems [3].

6.1 Behavioral Match

The match process argument in favor of an agent based solution can be expressed by the fact that an agent is able to deal with the aforementioned actions related to the control loop (see figure 1 §2.1). The term *agent* in computing covers a wide range of behavior and functionality. In general, an agent is an active computational entity that can perceive (through sensors), reason about, and initiate activities (through effectors) in his environment [12]. Normally, an agent has a repertoire of actions available to him. This set of possible actions represents his ability to modify his environment. The types of actions an agent can perform at a point of time include:

- Physical actions are interactions between agents and the spatial environment.
- Communicative actions are interactions between agents. They can be emission or reception actions.
- Private actions are internal functions of an agent. They correspond to an agent exploiting his internal computational resources.
- Decision action can generate communicative, physical and private actions. A decision action can also update the agent's beliefs.

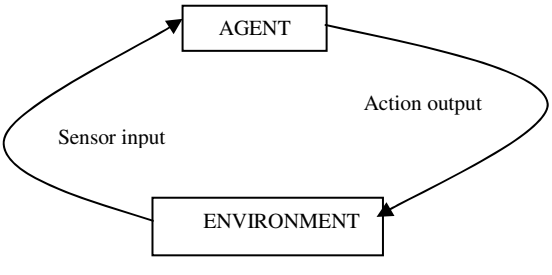

Fig. 6. An ongoing interaction between an agent and his environment

An action may be classified as either configuration, optimization, healing, or protection action depending on the reason of its execution. For example, substituting a candidate service is a physical action which can be optimization action if it is intended to guarantee a better QoS. It can be as well considered as a healing action in case of service failure.

6.2 Complexity Management

Computing systems consisting of software, hardware and communication infrastructure have become ever increasingly complex. If autonomic computing paradigm is to be engineered for such complex systems, hierarchical control architectures are considered as a key technology to rely upon [4, 13, 14]. In such case, a hierarchy of control loops is required to endow the whole system of self-managing mechanisms. Each level of control loop, achieving correspondingly, one of the different control goals which collectively constitute the overall control objectives of the system. Consequently, a multi-agent system solution is envisioned to deal with the self-* capabilities within an autonomic computing system.

For example, numerous autonomic managers in a composite Web service system must work together to deliver autonomic computing to achieve common goals. This is the case of a composite Web service which needs to work with the autonomic managers of the elementary Web services, registries in order for the Web service infrastructure as a whole to become a self-* system.

The argument in favor of a multi-agent system solution can also be described in terms of the ability of such systems to deal with complexity management. Previously, Booch identified three techniques for tackling complexity in software: *decomposition*, *abstraction* and *organization* [15].

- *Decomposition*: the process of dividing large problems into smaller, more manageable chunks each of which can then be dealt with in relative isolation.
- *Abstraction*: the process of defining a simplified model of the system that emphasizes some of the details or properties, while suppressing others.
- *Organization*: the process of identifying and managing the interrelationships between the various problem solving components. This helps designers tackle complexity in two ways. Firstly, by enabling a number of basic components to be grouped together and treated as a higher-level unit of analysis (e.g., the individual components of a subsystem can be treated as a single coherent unit by the parent system). Secondly, by providing a means of describing the high-level relationships between various units (e.g., a number of components may cooperate to provide a particular functionality).

Next, we deal with each technique in turn.

- *Agent-oriented decomposition is an effective way of partitioning the problem space of a complex system*: the agent-oriented approach advocates decomposing problems in terms of autonomous agents that can engage in flexible, high-level interactions. Decomposing a problem in such a way helps the process of engineering complex systems in two main ways. Firstly, it is simply a natural representation for complex systems that are invariably distributed and that invariably have multiple loci of control. This decentralization, in turn, reduces the system's

control complexity and results in a lower degree of coupling between components. The fact that agents are active entities means they know for themselves when they should be acting and when they should update their state. Such self-awareness reduces control complexity since the system's control know-how is taken from a centralized repository and localized inside each individual problem solving component [12]. Secondly, since decisions about what actions should be performed are devolved to autonomous entities, selection can be based on the local situation of the problem solver. This means that the agent can attempt to achieve its individual objectives without being forced to perform potentially distracting actions simply because they are requested by some external entity. The fact that agents make decision about the nature and scope of interactions at run-time makes the engineering of complex systems easier. Indeed, the system's inherent complexity means it is impossible to know a priori about all potential links: interactions will occur at unpredictable times, for unpredictable reasons, between unpredictable components. For this reason, it is futile to try and predict or analyze all the possibilities at design time. Rather, it is more realistic to endow the components with the ability to make decisions about the nature and scope of their interactions at run-time. Thus agents are specifically designed to deal with unanticipated requests and they can spontaneously generate requests for assistance whenever appropriate.

- The key abstractions of agent-oriented mindset are a natural means of modeling complex systems: In the case of a complex system, the problem to be characterized consists of subsystems, subsystems components, interactions and organizational relationships. Taking each in turn: firstly, there is a strong degree of correspondence between the notions of subsystems and agent organizations. They both involve a number of constituent components that act and interact according to their role within the larger enterprise. Secondly, the interplay between the subsystems and between their constituent components is most naturally viewed in terms of high level social interactions (e.g., agent systems are described in terms of "cooperating to achieve common objectives" or "negotiating to resolve conflicts"). Thirdly, complex systems involve changing webs of relationships between their various components. They also require collections of components to be treated as a single conceptual unit when viewed from a different level of abstraction. On both levels, the agent-oriented mindset again provides suitable abstractions. A rich set of structures is typically available for explicitly representing and managing organizational relationships such as roles (see [16, 17] for example). Interaction protocols exist for forming new groupings and disbanding unwanted ones (e.g., Sandholm's work [18]). Finally, structures are available for modeling collectives (e.g., teams [19]).
- *The agent-oriented philosophy for dealing with organizational relationships is appropriate for complex systems*: organizational constructs are first-class entities in agent systems. Thus explicit representations are made of organizational relationships and structures. Moreover, agent-based systems have the concomitant computational mechanisms for flexibly forming, maintaining and disbanding organizations. This representational power enables agent-oriented systems to exploit two facets of the nature of complex systems. Firstly, the notion of primitive component can be varied according to the needs of the observer. Thus, at one level, entire subsystems can be viewed as singletons, alternatively, teams

or collections of agents can be viewed as primitive components, and so on until the system eventually bottoms out. Secondly, such structures provide a variety of stable intermediate forms that, as already indicated, are essential for the rapid development of complex systems. Their availability means individual agents or organizational groupings can be developed in relative isolation and then added into the system in an incremental manner. This, in turn, ensures there is a smooth growth in functionality.

6.3 Pragmatic Reasons

Autonomic computing denotes a move from the pursuit of high speed, powerful computing capacity to the pursuit of self-managing mechanisms of computing systems. Indeed, today's computing and information infrastructure have reached a level of complexity that is far beyond the capacity of human system administration. For instance, follow the evolution of computers from single machines to modular systems to personal computers networked with larger machines. Along with that growth has came increasingly sophisticated architectures governed by software whose complexity now routinely demands tens of millions of lines of codes. The internet adds yet another layer of complexity by allowing us to connect this world of computers and computing systems with telecommunications networks. In the process, the systems have become increasingly difficult to manage, and ultimately, to use. Inspired by the functioning of the human nervous system which frees our conscious brain from the burden of dealing with some vital functions (such as governing our heart rate and body temperature), autonomic computing is considered as a promising solution to such problems.

As yet, however there is not a successful solution to autonomic computing which can be applied on a significant scale. So far a mature solution has not yet appeared. In part, this is due mainly to the youth of this paradigm and the absence of adequate tools, but our experience suggests that the absence of tools that allow system complexity to be effectively managed is a greater obstacle.

Agent technology is one of the most dynamic and exciting areas in computer science today. Many observers believe that agents represent the most important new paradigm for software development since object-orientation. Agent technology has found currency in diverse applications domains including ambient intelligence; grid computing where multi-agent system approaches enable efficient use of the resources of high-performance computing infrastructure in science, engineering, medical and commercial applications; electronic business, where agent-based approaches support the automation of information-gathering activities and purchase transactions over the internet; the semantic web, where agents are needed both to provide services, and to make best use of the resources available, often in cooperation with others ; and others including resource management, military and manufacturing applications [20].

Agent paradigm has achieved a respectable degree of maturity and there is a widespread acceptance of its advantages: a relatively large community of computer-scientists which is familiar with its use now exists. A substantial progress has been made in recent years in providing a theoretical and practical understanding of many aspects of agents and multi-agent systems [21].

7 Conclusion and Future Work

In this paper we adopt autonomic computing paradigm to propose an approach for self-adaptive web services. The basic idea is to consider web services as autonomic systems, that is, systems able to manage and adapt themselves to the changing environment according to a set of goals and policies. As a result, autonomic web services can recover from failure, optimize their performance, configure themselves, etc. without any human intervention. An autonomic distributed architecture is proposed where each component service is associated with one or a set of specific autonomic manager(s). We have also presented in this paper the main reasons to consider agent technology as a suitable candidate to deal with the technical machinery achieving self-adaptation within a Web services system.

Motivated by the fact that self-adaptation systems are recently considered as the trend of the new systems, and by the justified claim that agent-based computing has the potential to be integrated into the framework of autonomic computing we will show, in our future work, how software agents may be used to deal with autonomic Web services systems. More precisely, we envision to develop an agent-based architecture for an autonomic Web services system. An adaptive version of the *Search for Music* scenario, presented in this paper, is currently being implemented by using an agent based approach.

References

1. Ganek, A.G., Corbi, T.A.: The Dawning of the Autonomic Computing Era. J. IBM Systems 42(1), 5–18 (2003)
2. IBM Group.: An Architectural Blueprint for Autonomic Computing, `http://www-03.ibm.com/autonomic/pdfs/AC`
3. Chainbi, W.: Agent Technology for Autonomic Computing. J. Transactions on Systems Science and Applications 1(3), 238–249 (2006)
4. Tianfield, H., Unland, R.: Towards Autonomic Computing Systems. J. Engineering Applications of Artificial Intelligence 17(7), 689–699 (2004)
5. Console, L., Fugini, M.: The WS-Diamond Team: WS-DIAMOND: an Approach to Web Services - DIAgnosability, MONitoring and Diagnosis. In: e-Challenges Conference, The Hague (2007)
6. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL Processes with Dynamo and the JBoss Rule Engine. In: International Workshop on Engineering of Software Services for Pervasive Environments (ESSPE 2007), pp. 11–20 (2007)
7. Pernici, B., Rosati, A.M.: Automatic Learning of Repair Strategies for Web Services. In: 5th European Conference on Web Services (ECOWS 2007), pp. 119–128 (2007)
8. Brogi, A., Popescu, R.: Automated Generation of BPEL Adapters. In: International Conference on Service Oriented Computing (2006)
9. Ardagna, D., Comuzzi, M., Mussi, E., Pernici, B., Plebani, P.: PAWS: A Framework for Executing Adaptive Web-Service Processes. J. IEEE Software 24(6), 39–46 (2007)
10. Chafle, G., Dasgupta, K., Kumar, A., Mittal, S., Srivastava, B.: Adaptation in Web Service Composition and Execution. In: International Conference on Web Services, pp. 549–557 (2006)

11. Narendra, N.C., Ponnalagu, K., Krishnamurthy, J., Ramkumar, R.: Run-Time Adaptation of Non-functional Properties of Composite Web Services Using Aspect-Oriented Programming. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 546–557. Springer, Heidelberg (2007)
12. Jennings, N.: On Agent-based Software Engineering. J. Artificial Intelligence 117(2), 277–296 (2000)
13. Albus, J.S., Meystel, A.M.: Engineering of Mind: an Introduction to the Science of Intelligent Systems. Wiley, New York (2001)
14. Tianfield, H.: Formalized Analysis of Structural Characteristics of Large Complex Systems. J. IEEE Transactions on Systems, Man and Cybernetics. Part A: Systems and Humans 31(6), 59–572 (2001)
15. Booch, G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, Reading (1994)
16. Baejis, C., Demazeau, Y.: Organizations in Multi-Agent Systems. Journées DAI, Toulouse (1996)
17. Fox, M.S.: An Organizational View of Distributed Systems. J. IEEE Transactions on Systems, Man and Cybernetics 11(1), 70–80 (1981)
18. Sandholm, T.: Distributed Rational Decision Making. Multi-Agent Systems. MIT Press, Cambridge (1985)
19. Tambe, M.: Toward Flexible Teamwork. J. Artificial Intelligence Research 7, 83–124 (1997)
20. Luck, M., McBurney, P., Preist, C.: Agent Technology: Enabling Next Generation Computing. In: AgentLink II (2003)
21. D'inverno, M., Luck, M.: Understanding Agent Systems, 2nd edn. Springer, Heidelberg (2004)