A Decentralized Model for Self-managed Web Services Applications

José Mª Fernández de Alba, Carlos Rodríguez, Damiano Spina, Juan Pavón[1], and Francisco J. Garijo[2]

[1] Dep. Ingeniería del Software e Inteligencia Artificial, Universidad Complutense de Madrid
Facultad de Informática, Ciudad Universitaria s/n, 28040, Madrid, Spain
jpavon@fdi.ucm.es
[2] Telefónica Investigación y Desarrollo
Emilio Vargas, 6, 28043, Madrid, Spain
fgarijo@tid.es

Abstract. Self-management in distributed systems is a way to cope with the growing complexity of these ones today, and its support in existing systems requires a transformation in their architectures. This work presents a decentralized model for the implementation of self-management capabilities, which also has the advantage of avoiding the single point of failure (SPOF) issue, providing more robustness to the management system. The proposed architecture has been validated in a real distributed application.

Keywords: self-management, autonomic computing, multi-agent system.

1 Introduction

Autonomic Computing is a concept initially developed by IBM [1], with the intention to cope with the increasing complexity of systems, as they grow in the number of elements and information. This solution intends to automate many of the functions associated with computing. In concrete, as [1] specifies, a computing system has the autonomic capability if it can manage itself only with the high-level objectives from administrators. The goal of such *self-management* ability is to free system administrators from the details of system operation and maintenance while providing users with a high-performance service.

The four main aspects of self-management are: self-configuration (automatic seamless configuration parameter adjustment), self-optimization (automatic performance tuning), self-healing (automatic detection and reparation of problems) and self-protection (automatic prevention from attacks and cascading errors).

IBM proposed a layered architecture [11] in which the upper layers contain the Autonomic Managers (AMs), and the lowest layer is populated by the managed resources. The management interfaces of these resources are encapsulated as service endpoints, so that they can be accessed via an enterprise communication technology, like Web Services. The recommended management model is Web Service Distributed Management (WSDM) standard [3]. The AMs in the control layer are cooperating agents [5], which achieve their management goals following high-level policies. They share a knowledge source, which provide a common domain model and the high-level information.

J.M. Corchado et al. (Eds.): DCAI 2008, ASC 50, pp. 90–98, 2009.
springerlink.com

The WSDM specification [3] enables management-related interoperability among components from different systems and facilitates integration of new ones, improving scalability. It also provides mechanisms for proactively analyzing different component properties such as quality of service, latency, availability, etc. In [14] is described an implementation example which is based on the IBM approach using a centralized architecture with a common Knowledge Repository.

Other self-management architectures like RISE [12] are domain-specific. They focus on particular system aspects such as: image management [12], workflow adaptation [13] and pervasive computing [15].

The work presented in this paper proposes a framework for incorporating self-management capabilities into Web Services applications based on WSDM model. It provides Web Services and Web Applications with autonomous features such as fault diagnosis, dynamic rebinding, file restoring, and resource substitution.

The approach consists on making each WS component of the system Self-Managed. Instead of having a common Knowledge Repository, which is often a Single Point Of Failure (SPOF), each self-managed Component has self-knowledge about its own dependences, and social knowledge about their dependent components. The aim of the paper is to describe the proposed approach, which is illustrated with a working example of self-healing. The validation framework is based on a website supporting a distributed social network for artists.

The paper begins with the architectural approach presented in section 2. A more detailed description of this architecture is shown in section 3, focusing the planning model in section 4. The case study and the validation of the proposed approach are in section 5. Finally, a summary of the work done and future work are discussed in the conclusions at section 6.

2 Approach for Enabling Self-management Features

The proposed approach focus on distributed systems based upon Web Services technology. These systems could be transformed into self-management systems by applying a self-management framework to each component. The basic idea is to make each system component (WS) self-managed, by enhancing them with new management components implementing self-management capabilities. Then make the self-managed components

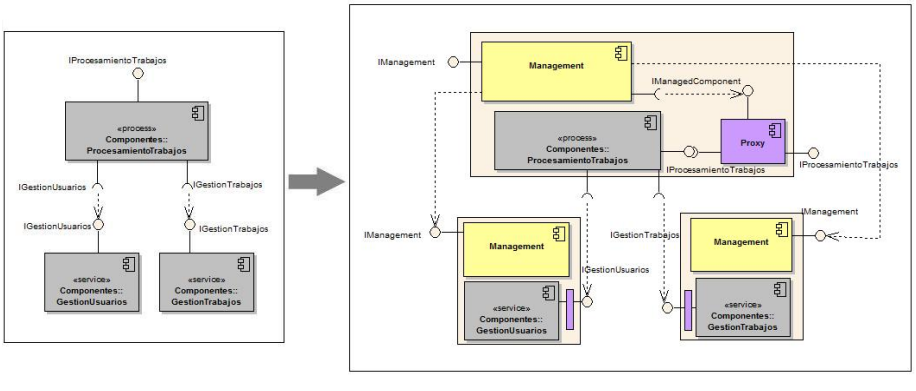

Fig. 1. Transforming a system into a self-managed system

cooperate in order to make the overall system self-managed. Figure 1, gives an example of transformation based on the studied case.

Each component has internal parts like files, libraries, etc., and possibly dependences with other components and servers. These components will be monitored, analyzed and controlled to provide self-management capabilities for each component and the whole system.

3 Self-managed Architecture

Figure 2 illustrates the internal structure of a service component. The "Component" is the original component that provides the logical operation of the service. The "Management" and "ProxyOfControl" components implement the management capabilities and are added to build the "NewComponent", which now has the original logical operation and self-management capability.

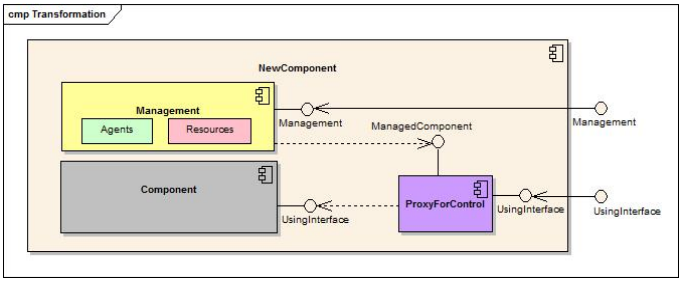

Fig. 2. Self-management component Architecture

The Management Component is made of packaged agents, resources and a model, which will be described later. The Management Interface offers operations to others self-managed components, which might use them to know its operational status.

The "ProxyOfControl" component controls the access to the managed component, avoiding possible misuses in inappropriate states, and providing information about the state of the managed component by catching Technical Exceptions. This component was designed using the State Design Pattern [10].

3.1 Modelling Dependencies

Achieving self-management capabilities require a conceptual model of the domain involved representing explicitly the dependencies among components [2]. Figure 3 shows the model of dependencies shared by the management components.

A managed component could have internal or external dependences: an internal dependence might be a dependence with a computing entity such as a file or a library, while an external dependence might be a dependence with a server, e.g. an email server, a database server, a file server or any application service. The application service external dependence refers to an abstract service, it means, a required interface, which is resolved at runtime.

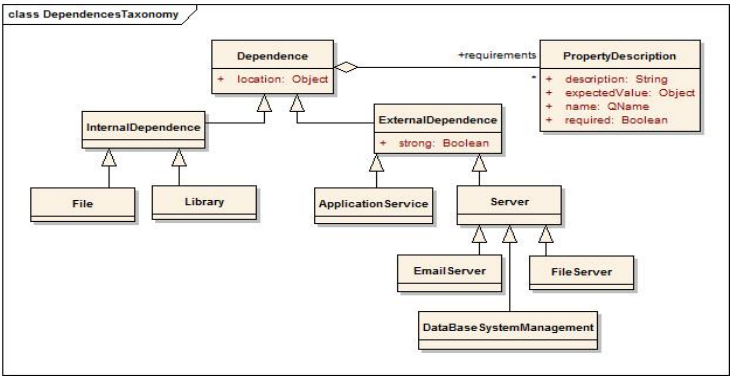

Fig. 3. Dependences

All dependences have a location. The location is an attribute indicating how to access the component that supplies the dependence, for usage or monitoring purpose. For application services, the location refers to the required interface, and the registry service location to find out a particular service to supply the dependence.

The dependence also has a set of requirements that define the properties to be satisfied by the related component.

A property description has the following attributes:

- Name: a full-qualified name.
- Description: a human-readable description.
- Required: if it is required or optional.
- Expected value: the expected value of the property.

Examples of properties are: can read, can write, XML well-formed, syntax of content validated, file names patterns, availability, time of response, etc.

3.2 Self-management Agents and Resources

The logical control was designed using the Multi-Agents paradigm [5], and it is implemented using component patterns based on the ICARO-T framework [6] [7]. There are four types of agents:

- **The Manager:** It is responsible for creating, terminating and management the rest of agents and resources in the Management Component.
- **The Installer:** It is responsible for verifying the internal dependences and to fix possible errors that may happen during the first execution of the managed component.
- **The Runtime agent:** It is responsible for verifying the right functioning of external dependences at runtime, passing control to the Repair agent when errors occur.
- **The Repair agent:** It has the responsibility of fixing errors send by the Runtime agent.

Resources perform different task required by the agents. Some of them are responsible for monitoring the properties of the managed component's dependences.

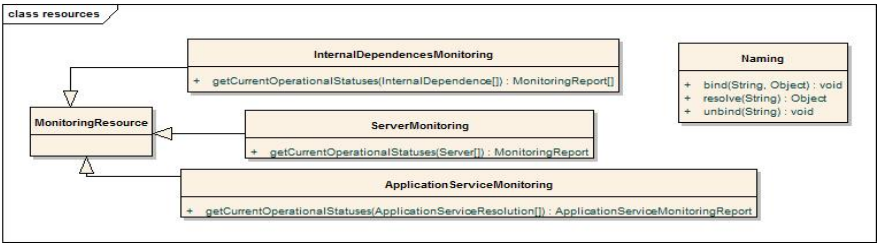

Fig. 4. Monitoring Resources

"InternalDependenceMonitoring" resource is in charge of getting the properties values of each managed component's internal dependence, and of inferring its operational status.

The "ServerMonitoring" resource is responsible for monitoring the servers as File servers, Database servers, etc, which are used by the managed components.

The "ApplicationServiceMonitoring" resource is responsible for monitoring application services used by the managed component. It monitors specific services instead of abstract services. It generates reports containing the service resolution of the abstract service dependence.

Monitoring resources generate reports that are read by agents to get the operational status of both internal dependencies of managed components, and external dependencies of those components. The Information about what to monitor is provided by the two XML description files: the Internal Dependence Description File (IDDF), and the External Dependence Description File (EDDF).

Agents use the Resources to monitor and analyze the internal structure of the managed component. The monitoring of external components is performed through queries and publish/subscribe mechanisms. Agents also gather information about the Managed Component state from the "ProxyOfControl". This information is used to achieve fault diagnosis.

3.3 The Behaviour of a Self-managed Component

The computing behaviour of a self-managed component will be illustrated with a working self-repair example taken from the Artist Community system, which has been used for validating the approach. The scenario is based on the failure of one running components –"GestionUsuarios"–, which affects the component "ProcesamientoTrabajos" depending on it. The system behaviour is depicted in figure 5. The Runtime Agent in "ProcesamientoTrabajos" detects the possible malfunction through its monitoring capability.

The Runtime Agent publishes the inferred status and stops the Managed Component "ProcesamientoTrabajos" because repair is needed. Then, it requests to the Manager to create an instance of the Repair Agent, which will be in charge of solving the problem. This agent first elaborates a repair plan in order to rebind an alternative of "GestionUsuarios", and then instantiates and executes the plan.

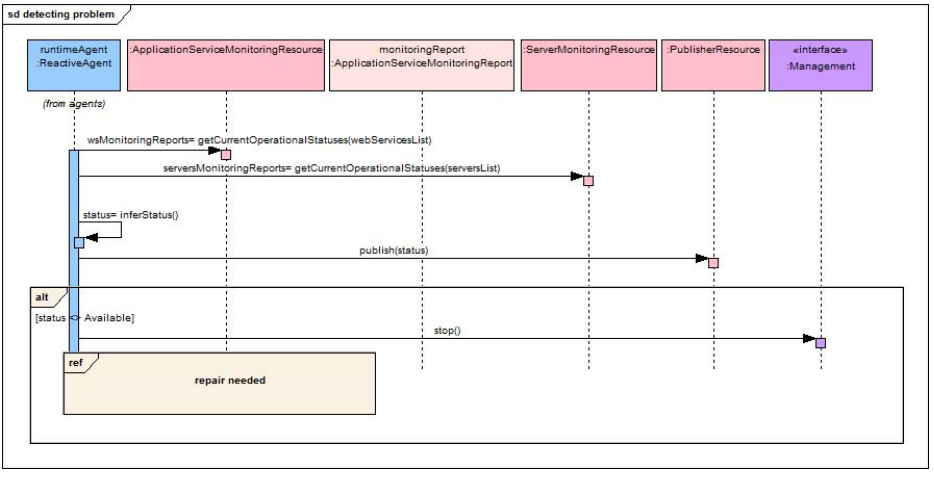

Fig. 5. A repair case

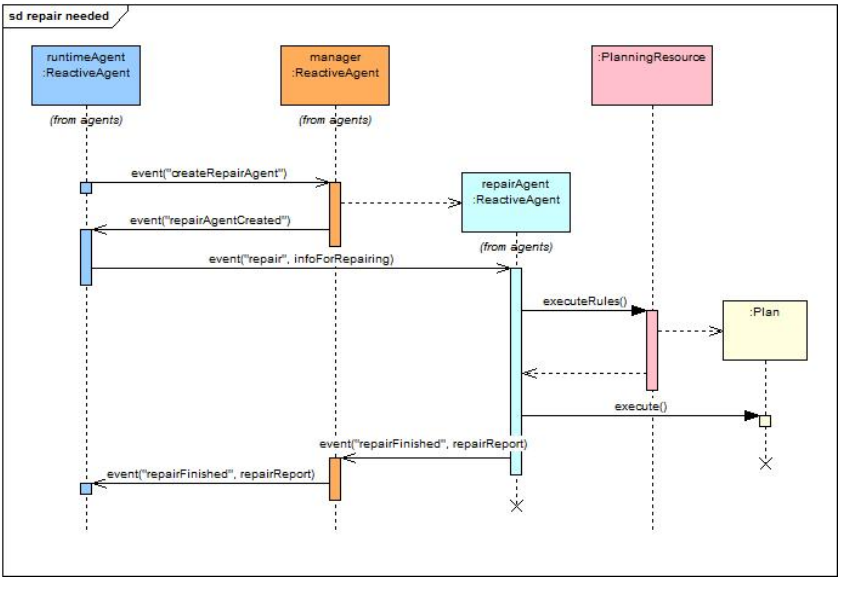

Fig. 6. Creation and execution of a plan by the Repair Agent

The repairing plan successes because an alternative service is available in the system. The new service rebound by the execution of the plan will be monitored in the next cycles of the Runtime Agents. If the new service's status is Available, the Runtime Agent will infer an Available status for the managed component, start it and publish the new status.

4 Planning Model

A Plan in this model is a sequence of Tasks. A Task is defined as an operator that somehow changes the environment state pursuing some objective.

The preparation of a plan consists in chaining different tasks in sequence. This process is dynamically performed by an agent anytime it detects some issue reported by monitoring resources with the intention to solve the problem.

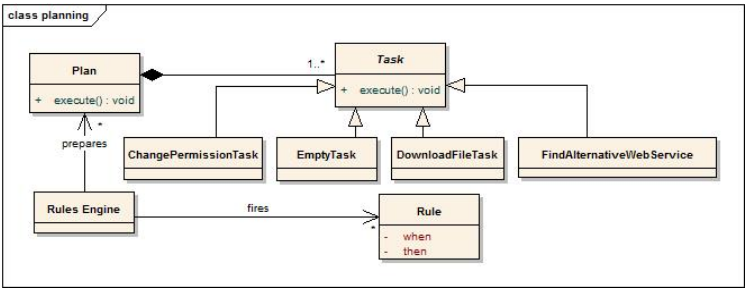

Fig. 7. The Planning Model

For an agent to decide which tasks are included in the plan, a set of "when-then" rules whose "when" part contains the possible symptoms detected for the possible issues. These rules are defined in a text file and fired by a rules engine based on RETE algorithm [9]. A rule example is given in figure 8. The set of predefined tasks and the rules file can be extended in order to customize the agents' behaviour against some issue.

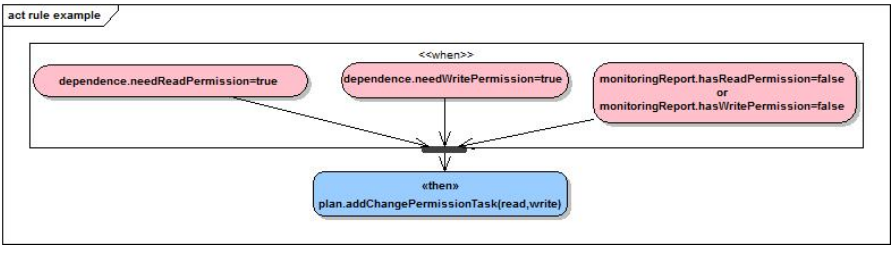

Fig. 8. A rule example

The preparation of the plan is finished when there are no more rules to fire. The plan is then ready to be executed, usually by the agent that prepared it.

5 Validation

The framework has been validated building a distributed system for assisting a Graphic Arts Community (the users) and then enhancing each system component with self-management capabilities.

The system is made of separated components that perform the different functions, some of them requiring others to their own functionality. The system is implemented using Java™ language and JAX-WS framework to support remote access via Web Services technology. Their interfaces and Access Points are registered in a central UDDI Registry. In addition, the system uses a SMTP Server, a Database Server and a UDDI Registry Server.

The transformation framework is made of a set of classes and file resources implemented with Java™, which are included together with business classes to generate a unique deployable component that runs on the same platform.

After framework application, the system has been successfully tested with a collection of significant scenarios including: restoration of missing files, XML validations, rebinding of services with replicated Web Services, etc.

Results showed that, although the computational overload is perceptibly increased, user-system interactions are not affected, while service continuity and stability are significantly improved.

6 Conclusions

The results obtained with the prototype show that the self-managed components perform successfully local monitoring, dynamic plan synthesis, and plan execution for component troubleshooting. Coordination among components is also achieved for fault diagnosis and self-healing. Compared to other approaches based on hierarchical management structures, making each component self-managed enforces their autonomy for failure detection and problem solving, and peer-to peer communication among components provides robustness and fault tolerance at system level. Decentralized control has also well known shortcomings with respect to centralized approaches, as the need of more sophisticated protocols for communication and cooperation. However, for large systems the advantages overcome the disadvantages, because this kind of architecture avoids bottlenecks, are more flexible and can be easily extended.

Future work should focus on self-optimization, self-configuration and self-protection. The last objective could be achieved following a similar approach consisting on enhancing each component with self-protection capabilities. This idea may not be applicable to the first two objectives. Achieving self-optimization and self-configuration would require system-wide parameter adjustment based on global system features that must be obtained seamlessly from system components. Therefore, individual components should "agree" on the proposed changes to achieve the global system behaviour. This might be done through the introduction of component's *choreographies* –group tasks carried out by agents in order to achieve common objectives, which are supported by some interaction protocols.

Another key issue is the automatic generation of Proxy classes and configuration files from code annotations made by developers in the business component code. This might be done by developing specific tools that will interpret the code annotations to detect component's usage of Web Services and other external components, as well as internal dependences. This annotation-oriented dependence declaration style, seems more intuitive and less error-prone than hardcoding dependency description files.

Finally, the self-management framework can be applied to itself since it is also a system. This can be useful to prevent errors in management tasks and to ensure that the machinery (configuration files and auxiliary classes) is ready.

Acknowledgements. We gratefully acknowledge Telefónica I+D, and the INGENIAS group for their help and support to carry out this work.

References

1. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer Magazine on January, 41–50 (2003)
2. D'Souza, D.F., Wills, A.C.: Objects, Components and Frameworks With UML. Addison Wesley, Reading (1999)
3. OASIS WSDM Standards, An Introduction to WSDM, http://docs.oasis-open.org/wsdm/wsdm-1.0-intro-primer-cd-01.pdf
4. OASIS WSDM Standards, Management Using Web Services Part 2, http://docs.oasis-open.org/wsdm/wsdm-muws2-1.1-spec-os-01.pdf
5. Mas, A.: Agentes Software y Sistemas Multi-Agentes: Conceptos, Arquitecturas y Aplicaciones
6. Garijo, F.J., Bravo, S., Gonzalez, J., Bobadilla, E.: BOGAR_LN: An Agent Based Component Framework for Developing Multi-modal Services using Natural Language. In: Conejo, R., Urretavizcaya, M., Pérez-de-la-Cruz, J.-L. (eds.) CAEPIA/TTIA 2003. LNCS (LNAI), vol. 3040, p. 207. Springer, Heidelberg (2004)
7. The ICARO-T Framework. Internal report, Telefónica I+D (May 2008)
8. IBM, An architectural blueprint for autonomic computing, section 2.2
9. Forgy, C.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence 19, 17–37 (1982)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software
11. IBM, An architectural blueprint for autonomic computing
12. Lee, J., Jeong, K., Lee, H., Lee, I., Lee, S., Park, D., Lee, C., Yang, W.: RISE: A Grid-Based Self-Configuring and Self-Healing Remote System Image Management Environment. In: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing (e-Science 2006) (2006)
13. Lee, K., Sakellariou, R., Paton, N.W., Fernandes, A.A.A.: Workflow Adaptation as an Autonomic Computing Problem. In: WORKS 2007 (2007)
14. Martin, P., Powley, W., Wilson2, K., Tian, W., Xu1, T., Zebedee, J.: The WSDM of Autonomic Computing: Experiences in Implementing Autonomic Web Services. In: International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2007) (2007)
15. Ahmed, S., Ahamed, S.I., Sharmin, M., Haque, M.M.: Self-healing for Autonomic Pervasive Computing. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876. Springer, Heidelberg (2007)