Achieving Software Robustness via Multiagent-Based Redundancy (Extended Abstract)

Rosa Laura Zavala Gutiérrez and Michael N. Huhns
University of South Carolina,
Department of Computer Science and Engineering
Columbia, SC 29208 USA
{huhns, zavalagu}@engr.sc.edu

Abstract

Conventional methods for software development, testing and validation do not provide the levels of reliability and robustness that consumers of large, complex and heterogeneous systems will demand. This document introduces the idea of multiagent-based redundancy for robust software development. Robustness can be achieved through redundancy, and we hypothesize that agents by being naturally smaller and easier to program than conventional systems, are an appropriate unit for adding redundancy. Agents having different algorithms but similar responsibilities produce the redundancy. The paper presents a testbed related to autonomic computing that is based on the use of existing Web services. We discuss the results obtained from the testbed. We also discuss the applicability of the work presented here to autonomic computing. We present in a nutshell results obtained from some other experiments we have run so far, as well as our future research plans.

1 Introduction

Making robust software systems has been always an issue of importance in software engineering. It is a highly desirable and sometimes indispensable requirement that a system be able to function correctly or coherently in a changing environment, in the presence of invalid or conflicting inputs, and in the presence of situations not considered during its design.

The complexity of systems has increased significantly in recent years. Computer systems are now entrusted with control of global telecommunications, electric power distribution, water supplies, airline traffic, weapon systems, and the manufacturing and distribution of goods. Such tasks are typically complex, involve massive amounts of data, affect numerous connected devices, and are subject to the uncertainties of open environments such as the Internet. Our society has come to expect uninterrupted service from these systems. Unfortunately, making robust software systems becomes a more challenging task as the complexity of the systems increases: it is extremely difficult and sometimes impossible to anticipate all the possible scenarios in which these complex systems will perform so as to make the appropriate tests.

The complexity of current computing systems as well as their robustness is also an issue addressed in the field of Autonomic Computing. Large, complex and heterogeneous systems as seen by autonomic computing [Kephart and Chess, 2003; IBM, 2002] will be very difficult to test effectively, and to deploy with the levels of confidence that consumers will demand: "Testing autonomic elements and verifying that they behave correctly will be particularly challenging in large-scale systems because it will be harder to anticipate their environment, especially when it extends across multiple administrative domains or enterprises" [Kephart and Chess, 2003]. Nevertheless, they have to be robust in order to make things simpler for administrators and users of IT.

This document introduces the idea of multiagent-based redundancy for the development of robust software, even conventional, non-agent-based software. Robustness can be achieved through redundancy, and we hypothesize that agents are an appropriate unit for adding redundancy. By being naturally smaller and easier to program than conventional systems, cooperative and communicative, able to allow dynamic composability and interaction abstractions, and able to represent multiple viewpoints, negotiate and use different decision procedures, agents seem to be the best approach to incorporate software redundancy. The agents have different algorithms but similar responsibilities.

In the next section we summarize the methodology we have been using. In section three we present a testbed related to Autonomic Computing that is based on the use of existing Web services. Then, we present results that show an improvement in robustness due to redundancy. We conclude by discussing our future research plans.

2 Methodology

Our goal is to create robust software systems and we propose to do that through massive redundancy, where the redundancy is managed by techniques developed for multiagent systems. That is, agents will represent the individual algorithms and components, and will use tech-

niques for cooperation and negotiation to achieve coherent, system-wide behavior.

Several researchers have investigated the use of multiagent systems for the development of software systems. Jennings has shown that multiagent systems can form the fundamental building blocks for software systems, even if the software systems do not themselves require any agent-like behaviors [Jennings, 2000]. When a conventional software system is constructed with agents as its modules, it can exhibit several additional benefits [Coelho *et al*, 1994; Huhns, 2001].

Because agents can represent multiple viewpoints and can use different decision procedures, they can produce more robust systems. The essence of multiple viewpoints and multiple decision procedures is redundancy, which is the basis for error detection and correction.

We aim to create agent systems in which the agents work together as a group and give better results than those they give as individuals. We construct the agent systems out of software components and wrappers for those components, which give them core agent grouping capabilities, that is, the capabilities needed to participate in a group decision. Each agent in the system has different software components but similar responsibilities. Its agent-based wrapper knows nothing about the inner workings of the component. It has knowledge only about the characteristics of its component, such as the input data type, the output data type, its time complexity and its space complexity.

The methodology we are investigating requires many diverse copies of algorithms and components. A problem is how to obtain these. One solution is to rely on open-source communities of developers. Another potential source of algorithms can be based on Web services, which is the option elaborated in this paper.

For combining their functionalities, the agents jointly agree on a solution. We studied possible modes for the agents to do that [Huhns *et al*, 2003]. A preprocessing approach would consist of the agents choosing, at the beginning, which one or ones are going to perform the task. Techniques for this approach include randomly picking an agent, selecting an agent by auction or voting (using information such as reliability and past performance of components), and distributing the task to be performed into subtasks to individual agents. A postprocessing approach would consist of all the agents performing the task and then deciding on which one produced the best result. Techniques for this approach include taking the result of the agent whose processing was the fastest, choosing the result given by most agents (voting), making a decision only about controversial data subsets, and incremental voting. A combination of the preprocessing and postprocessing approaches could also be used.

A wider description of these techniques as well as additional interaction protocols for the coordination and communication among agents when making a decision can be found in [Huhns *et al*, 2003]

3 A Testbed for Autonomic Computing

We created client agents for a number of Web services for weather, each offered by a different provider. Currently we have clients for ten different weather Web services. Then we converted each component or Web service into an agent by wrapping it. The agent gives to the component the capabilities to interact with others in order to jointly agree on a solution.

Each agent consists of a component (weather Web service) and a wrapper for that component. The wrapper knows nothing about the inner workings of the component. It only knows about the external characteristics of its component, such as its input and output data type(s), its location, the methods that it exposes (its interface) and its complexity. The agent wrappers were written in JADE and make use of the Java JAX-RPC and SAAJ APIs, as well as the Apache AXIS SOAP implementation. The agents can be viewed as SOAP clients with agent capabilities, i.e., protocols for handling communication, negotiation, and interaction.

A distributed preprocessing approach was used for an initial experiment on our testbed [Zavala, 2003]. The agents use the FIPA-request protocol to communicate and agree on a solution. Upon a user request, each agent decides, based on the input data type and desired output, whether its component is capable of executing the user request. All the agents whose component is capable of executing the user request try to run it and the first valid result (does not return null or a fault) is drawn. In this way, over a period of time, the fastest and most available components are chosen the most often, though some connections are very slow or sometimes not available.

Together, our weather agents provide a more robust weather service than any individual weather Web service alone: they work faster, have less connection failures (is much more unlikely that none of the ten connections work), and they work for a wider set of scenarios. Some of them accept only a US zip code as input. Others give the weather for different countries, i.e., one of them works only for Iraq (asking for the name of the city) and US (asking for the zip code) while another can give the weather for almost every country by means of an airport code. Some of them give temperatures in Fahrenheit, others in Celsius, and others in both scales. Also, some components provide only the current temperature, while others provide additional information like *n*-day forecasts, humidity, sky, wind, visibility, location, etc. The components vary also on the additional methods provided to make their use easier, i.e., list of the countries for which they can give the weather, list of airports in a particular country or region, list of regions in a country, etc. Finally, some connections are faster than others as well as steadier.

In earlier experiments we built three agent systems, each for a different domain (sorting a list of elements, reversing the order of the elements in a list, and evaluating arithmetic expressions defined in postfix notation). The collected software components or algorithms have

different characteristics, such as input data type, output data type, time and space complexity, performance and correctness. They were all written in Java and the wrappers in JADE. We conducted a set of experiments with the agent systems and with the software components alone. The architectures we used for the combination of the agent's functionality were a distributed pre-processing approach and a distributed post-processing approach. From our experiments we could observe that more algorithms produced better results than any one algorithm working alone. Together, they succeeded for a wider set of inputs than the inputs accepted by any individual component. Also, working together they avoided raising exceptions that were raised in some situations when working alone.

4 Conclusions and Future Work

Our research intersects with the realm of autonomic computing in several ways. First, our methodology can benefit from considering the autonomic elements necessary to facilitate redundancy through agents, such as self-monitoring, self-awareness, and environment-awareness. Second, autonomic computing applications can use our methodology at different levels: (a) making individual components of autonomic systems robust, which would help in the fault tolerance requirement of those components; and (b) customers could apply our approach by creating redundant agent systems out of services from different providers. That would make their software more robust. Finally, software robustness provided by multi-agent-based redundancy should alleviate the necessity of a self-healing task.

We are currently working on the development and use of ontologies to allow a better combination of components and knowledge sharing between the agents. The existence of a weather ontology, including concepts such as forecast, humidity, sky, wind, visibility, location, airport, scales, zip code, countries, regions, and cities, would allow the agents to compare their results and maybe combine them to provide a better service to a user. For example, if two agents that output the temperature in different scales (i.e., Fahrenheit and Celsius) were to compare their results they must be aware of that difference so that they do not mistakenly compare values having different units. Even more, they could convert them to an agreed scale.

Our interest is in experimentation on large-scale systems, i.e., wrapping agents around redundant software components written in different computer languages, different OSs, and distributed geographically. We will continue our development of the Web services testbed.

A more interesting solution is to imagine a range of developers from a broader class of our society. It is possible that through well programmed and verified agent wrappers, software of a variety of types from a variety of developers could be accommodated. Just as the Web enables a wide range of people to publish and distribute *information*, this would enable more people to develop and contribute *behavior*. The resultant systems of aggregated behavior, such as those for finances, electrical power distribution, and telecommunications whose behavior affects the lives and well being of the members of a society, would be more likely to operate on behalf of those members.

The field of machine learning, in particular wrapper learning, can be explored for cases where the collection of software components is vast and it is not viable to create a functional interface for all the components according to the format of the one needed by our wrapper. Wrapper learning techniques could be applied to make a wrapper generator module that would generate wrappers for each software component without the need of a functional interface for those components. This module would learn to deduce the information needed (input parameters, way of running the component) from either a functional interface provided by the component provider (which could be in any format, even natural language) or the code of the component (one module generator per language).

Just as there are a number of ways that a group of people can reach conclusions and make decisions, so are there a number of ways that a group of agent-wrapped software components can combine their results. We will study alternative ways for combining the agents' functionality, i.e., a combination of preprocessing and post-processing approaches, or a standby approach where redundant components in the system are not used until a primary service provider fails. Also, different techniques for each approach can be tested, i.e., instead of choosing the outcome reached by a majority of the agents in the preprocessing approach, we could use the average of the results (for the particular domain, such as weather).

Acknowledgments

The US National Science Foundation supported this work under grant number IIS-0083362.

References

[Coelho *et al*, 1994] Helder Coelho, Luis Antunes, and Luis Moniz, "On Agent Design Rationale," in *Proceedings of the XI Simpósio Brasileiro de Inteligência Artificial (SBIA)*, Fortaleza (Brasil), October 17-21, 1994, pp. 43-58.

[Huhns *et al*, 2003] Michael N. Huhns, Vance T. Holderfield, and Rosa Laura Zavala Gutierrez, "Achieving Software Robustness Via Large-Scale Multiagent Systems," in *Software Engineering for Large-Scale Multi-Agent Systems*, Alessandro Garcia, Carlos Lucena, Franco Zambonelli, Andrea Omicini, and Jaelson Castro, editors, Springer Verlag, Lecture Notes in Computer Science, Volume 2603, Berlin, 2003, pp. 199-215.

[Huhns, 2001] Michael N. Huhns, "Interaction-Oriented Programming," in *Agent-Oriented Software Engineering*,

Paulo Ciancarini and Michael Wooldridge, editors, Springer Verlag, Lecture Notes in AI, Volume 1957, Berlin, 2001, pp. 29-44.

[IBM, 2002] Autonomic Computing: IBM's Perspective on the State of Information Technology, available at http://www.research.ibm.com/autonomic/manifesto/, June 2002.

[Jennings, 2000] Nicholas R. Jennings, "On Agent-Based Software Engineering," *Artificial Intelligence*, Vol. 117, No. 2, 2000, pp. 277-296.

[Kephart and Chess, 2003] Jeffrey O. Kephart and David M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, Vol. 36, No. 1, January 2003, pp. 41-50.

[Zavala, 2003] Rosa Laura Zavala Gutiérrez, "Weather Agents Testbed." http://www.cse.sc.edu/~zavalagu/redundancy/testbed