Chapter 14

ON BUILDING ROBUST WEB SERVICE-BASED APPLICATIONS

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

Abstract Reliability is an issue of importance in Web service applications, as they often expose an enterprise's critical processes, such as finance, insurance, customer-relationship management, or sales. Reliability can be ensured by providing compensations for faults, but the complexity of the applications makes it impossible to anticipate all the scenarios that the applications might encounter. An approach to error tolerance needs to be provided. In this chapter we show that even when online applications are complex, the service-oriented architecture of Web services provides an excellent basis for our redundancy-based approach to error tolerance. Our studies and theories in software robustness demonstrate that: (i) robustness can be achieved through redundancy; (ii) by being able to represent and conciliate multiple viewpoints, agents are an appropriate unit for adding redundancy; and (iii) agents having different algorithms but similar responsibilities can produce the needed redundancy. We integrate Web services as part of the functionality of the agents that provide the redundancy.

1. INTRODUCTION

During the last decade, the software industry has rapidly evolved many paradigms, architectures, and technologies for developing enterprise-wide software applications. The applications have capabilities that were not possible or were very difficult to provide previously, such as integration of disparate applications within and among enterprises, automation of business-to-business (B2B) data interchange and negotiations, and definition and integration of business processes involving different parties. Initial B2B automation activities were tightly coupled, in the sense that business partners predefined the

terms of their interaction using standards such as EDI[1] and XML[2]. Recently, the emergence of Web services (WS[3]) has led the software industry into a service-oriented approach to software development. WS is a loosely coupled technology, because Web services are based on the autonomous use of standard protocols (UDDI[4] for discovery, WSDL[5] for description, BPEL4WS[6] for coordination, and SOAP[7] for communication). WS provide greater flexibility with respect to the interoperability, reuse, and development of applications in a distributed environment.

Reliability is an issue of importance in Web service applications, as they often expose an enterprise's critical processes, such as finance, insurance, customer-relationship management, or sales. However, those applications are complex and have so many related entities that it is impossible to anticipate all the scenarios that the applications might face. An approach to error tolerance needs to be provided.

To satisfy this need, we hypothesize that multiagent-based redundancy can provide a basis for robust WS-based applications, and a service-oriented approach for implementation and deployment can foster that. Developers can identify critical points in a process and incorporate redundant functionality based on existing Web services. Since Web services reside on a vendor site, they may be an affordable basis for redundancy, because consumers do not buy the component itself but only pay to access it. Further, they provide dynamic discovery and aggregation of services and, at any time, consumers may decide to no longer use a particular Web service.

However, if we use a Web service *per se* as the unit for adding redundancy, some centralized controller would have to be in charge of managing the execution of the redundant components and the selection of results. We further hypothesize that agents—by possessing the abilities to represent multiple viewpoints and interaction abstractions, negotiate, use different decision procedures, and reach agreements—are an appropriate unit for adding redundancy. Agents having different algorithms but similar responsibilities can produce the needed redundancy. We use Web services as the source for the agents' algorithms.

In Section 2, we provide a review of software reliability and software robustness. Section 3 includes a discussion of approaches to error tolerance for achieving software robustness. In Section 4, we present the details of our approach. Section 5 presents a description of the agent-based Web applications

[1] http://developer.netscape.com/viewsource/marchal_edata.htm
[2] http://www.w3.org/XML/
[3] http://www.w3.org/2002/ws/
[4] http://www.uddi.org/
[5] http://www.w3.org/TR/wsdl
[6] http://xml.coverpages.org/bpel4ws.html
[7] http://www.w3.org/TR/soap/

that we developed. Finally, we present our conclusions and recommendations for future work in Section 6.

2. BACKGROUND

An algorithm is said to be correct if, for every input instance, it halts with the correct output. An incorrect algorithm might not halt at all on some input instances, or it might halt with an answer other than the desired one (5). Even when the algorithm is correct, its implementation might not be correct (due to mistakes introduced by the programmer) or if it is, it could have additional constraints imposed by the programmer in order to make it easier to implement, or it could face unexpected situations (i.e., a faulty environment, corrupted data, or defective hardware).

Software robustness is the ability of a software product 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. Software reliability is a wider concept, which includes robustness. It is the probability of error-free operation of an application (16). The required level of reliability in a system depends on the consequences and cost of runtime errors. Safety-critical systems have very high reliability requirements, because errors during performance could have catastrophic consequences. For other systems, the cost of a small number of errors during performance may be acceptable. Even for non-critical software systems with lower reliability requirements, software reliability is an important attribute and has frequently been studied in software engineering. Reliability in a software system can be achieved using the following complementary strategies: (1) error prediction, (2) error avoidance, (3) error detection and correction, and (4) error tolerance.

Error prediction uses statistical techniques to estimate how many flaws might be in a system and how severe their effects might be. Based on that information, management can decide whether the statistics are acceptable or if other reliability strategies should be used to improve the software reliability. The use of good software engineering techniques and processes helps towards error avoidance. However, human errors cannot be avoided during development, resulting in bugs in the resultant software product. Procedures such as JUnit[8] testing can aid in detecting the bugs so that the developers correct them before releasing the product. The tests can also include scenarios not stipulated in the requirements, thus testing the software robustness. Even after testing, it is probable that a product will still contain bugs. Moreover, even if the software performs to its specification (it does not contain any bugs) and the tests consider

[8]http://www.junit.org/

exceptional cases, it is impossible to anticipate all the possible scenarios under which the system will operate. Therefore, we need to consider the fact that the software application will face unexpected situations.

Error tolerance is a strategy aimed at enabling a system to continue operation even in the presence of an unexpected situation. Thus, an approach to achieve error tolerance is required to produce robust software. At the present, there are few such approaches. Section 3 includes a discussion of the most common ones: N-version programming, recovery blocks, and transactions.

3. RELATED WORK

3.1 N-VERSION PROGRAMMING

In *N-version programming*, at least three different versions of a software system are implemented by different teams from a common specification. The different versions are executed in parallel. Their outputs are compared and the final result is determined by using a voting system. This approach is based on the idea that different teams will not make the same errors when designing and implementing a system (1).

3.2 RECOVERY BLOCKS

At least two different implementations for the same problem are collected. These are not based on the same specification. A test to check if the program has executed successfully has to be provided, which usually receives the program output and checks that it is correct according to the input provided. The different implementations are executed in sequence rather than in parallel. After one version is executed, the test to check correctness is run and the next version is executed only if the test fails (13, 14).

N-version programming and Recovery blocks technologies are based on the assumption that the specification is correct. The Recovery blocks technology also assumes that the program that tests correctness is correct.

3.3 DEFENSIVE PROGRAMMING

Database systems have exploited the idea of transactions for maintaining the consistency of their data. A transaction is an atomic unit of processing that moves a database from one consistent state to another. A similar approach to database transactions for general software applications is known as *defensive programming*. Defensive programming checks the system state after modifi-

cations to ensure consistent state changes. If inconsistencies are detected, the state is restored to a known correct state. Restoration of a state is achieved using one of two mechanisms: *backward recovery* (restore the system to a known correct state) and *forward recovery* (try to correct the damaged system state).

When a system includes cooperating processes, the sequence of process communications can be such that the check-points of the processes are out of synchronization. To recover from a fault, each process has to be rolled back to its starting state. This makes recovery very complex.

The three approaches presented in this section—recovery blocks, N-version programming, and defensive programming—require the use of a centralized controller to ensure that the steps involved in tolerating an error are executed (16).

4. MULTIAGENT-BASED REDUNDANCY USING WEB SERVICES

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 (11). When a conventional software system is constructed with agents as its modules, it can exhibit several additional benefits (4, 7).

Our goal is to create robust WS-based applications and we investigate its achievement through massive redundancy, where the redundancy is managed by techniques developed for multiagent systems. That is, agents represent the individual Web services and use techniques for cooperation and negotiation to achieve coherent, system-wide behavior.

Consider the real world application of credit approval. A credit report contains information about an individual's credit worthiness (i.e., payment history and any suits, arrests, or filings for bankruptcy). Companies called credit reporting agencies compile and sell credit reports to businesses. Because businesses use this information to evaluate applications for credit, insurance, employment, leasing, and other similar purposes, it is important that the information in the report be complete and accurate. Unfortunately, this is not always the case. Sometimes a credit report might not reflect all the individual's credit accounts: it might include payments not credited or data mixed in from the credit file of someone else with a similar name.

Figure 1 depicts the credit approval process as conducted by some particular business. The process starts when an application is submitted. Then, the business collects necessary information (name, address, phone number, employer's information, income, immigration status, etc.). Next, the business determines

the veracity of the information provided by the applicant. For example, it could call the applicant's employer to make sure that it is really her employer and to verify the applicant's income. After that, it is necessary to obtain the applicant's credit score, and it is at this point that the process can benefit from redundancy. The business can make use of multiple Web services provided by different parties, i.e., different credit reporting agencies, to obtain the applicant's credit score.

To see how additional services compensate for inconsistent information, imagine that an agency does not have all the applicant's credit accounts on its system. If the business consulted with only that agency, the inconsistency could result in a denial of credit due to "insufficient credit history." However, if the business consults with more agencies, then the inaccuracies of the first agency would become apparent. Additional services also compensate for an unavailable service: e.g., if a credit reporting agency's Web server is down, information can be collected from the other agencies. The use of multiple Web services provides the basis for an improved decision with respect to whether to approve or deny credit.

The challenge is to develop techniques for designing software systems so that the systems can easily accommodate the additional components and take advantage of their redundant functionality. In our example, the system that implements the credit approval process would need to include some decision mechanism that took the results from the different credit reporting agencies and provided the result. At first glance, this might seem an easy task—just compare results and if they differ apply a single rule for deciding which one to take—but it implies satisfaction of all the details of reaching an agreement among different parties.

If agents are developed at a convenient level of granularity at which to add redundancy, then the software environment that takes advantage of them is akin to a society of such agents, where there can be multiple agents filling each societal role. By design, agents know how to deal with other agents, so they can accommodate additional or alternative agents naturally. They are also designed to reconcile different viewpoints.

4.1 ARCHITECTURE AND PROCESS

We propose the following process to create robust WS applications:

- The developers of the application have to identify critical points in a process and collect redundant functionality for those processes based on existing Web services. The service-oriented approach of WS provides an excellent basis for that. Since Web services reside on the vendor site, they may be an affordable basis for redundancy, because consumers do

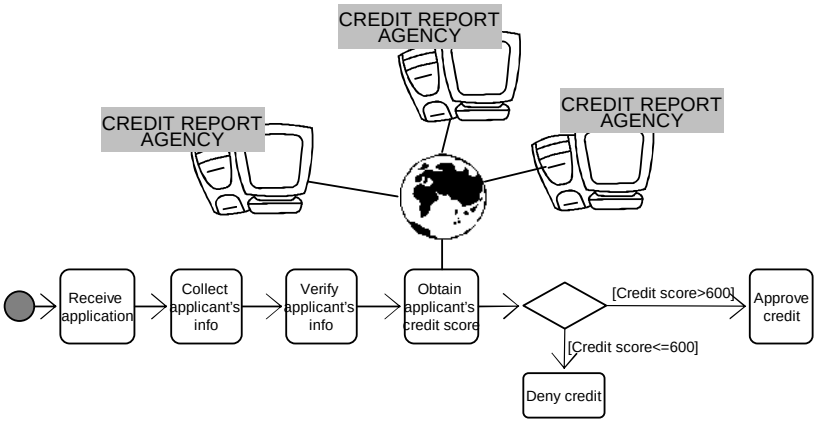

Figure 14.1. Redundancy is the basis for most forms of robustness

not buy the component itself, they just pay to access it. Further, they provide dynamic discovery and aggregation of services, and at any time, consumers may decide not to use a particular Web service any more.

- Develop the agents (once per Web service collected). The Web service will be part of the functionality of the agent, but it will also include the capabilities needed to participate in a group discussion and reach consensus. The agent should know nothing of the inner workings of the WS. It should have knowledge only about its characteristics, such as access point, input data type and output data type, and time and space complexity. For developing the agents we need to:
  - Choose the approaches that the agents will use for accommodating the redundant functionality. The possible approaches that we have defined (8) are explained in the rest of this section.
  - Integrate the agents and Web services so that the agents are accessed instead of the Web services. Integration of agents and Web services is an open and emerging research area (10). Efforts are underway to target the particular problem of agents using Web services as part of their behavior and wrapping Web services with agent capabilities. The focus of our paper is not centered on this point, so for the development of our example applications we used a simple approach, which is described in Section 5. Other approaches are described in (2, 6).

Possible modes that we have defined in order for the agents to reach consensus are listed in Table 1.

Table 14.1. Approaches for Combining Agents' Functionalities

PREPROCESSING	*POSTPROCESSING*
Random / lottery	Performance-based
Auction election / criteria selection	Voting
Team	Collaboration
	Incremental

A preprocessing approach would consist of the agents choosing, at the beginning, which one or ones are going to perform the task. There are three strategies applicable to this approach:

- *Randomly picking an agent to perform the task.* This is equivalent to a lottery. The output would be based solely on the results from that agent. Any bugs or errors in the agent would not be caught or corrected. The lottery method would be appropriate in a system where all agents have the same capabilities, or in a system with a relatively large number of correct agents and the probability of selecting an appropriate agent is high. Communication overhead would be low as it would be needed only for determining the winner of the lottery.

- *Selecting an agent by auction or voting (using information such as reliability and past performance of components).* Since this is a single-input, single-output sub-system, an agent's desire to perform a task would be based on mitigating factors the agent knows or can deduce about itself, such as speed, complexity, and reputation. These factors would help in determining which agent is chosen to perform each task. It would be the means for determining the agent's bid in an auction or the value (or weight) of an agent's vote in an election. This method, while also based on a single agent's response, is a more intelligent choice since justifying factors are involved in the selection. The domain is similar to that of the lottery method.

- *Distributing the task to be performed into subtasks to individual agents.* This strategy would entail distributing the task to be performed into subtasks to individual agents. The individual agents would be responsible for processing only a subset of the original task. The subsets would then be collected and combined to contribute to the single answer required by the system. This methodology would increase speed as far as the processor goes. If all agents are equally competent, then this method is practical for a large problem that could be divided into smaller subsets.

The problem of selecting an agent to perform a task, as well as distributing a task among different agents has been largely discussed in previous Distributed AI and Multi-Agent Systems literature, beginning with the Contract Net Protocol and extending through market approaches, auctions, and distributed planning (15, 3, 12).

A postprocessing approach would consist of all the agents performing the task, followed by a decision on which one produced the best result. There are four strategies applicable to this approach:

- *Taking the result of the agent whose processing was the fastest.* A domain in which the agents are sufficiently competent would be an appropriate for this strategy.

- *Choosing the result given by most agents by voting.* This is different from the voting scheme above, in that the proposed output would be based on a direct comparison of information. Agents would compare their results to other agents and a running tally kept. The result with the most votes is given as the final answer. To handle ties, a weight could be assigned to each agent based on additional factors such as speed and reputation.

- *Making a decision only about controversial data subsets.* This involved a collaboration strategy where data is compared between agents so that any common data subsets are kept and only a decision about controversial data subsets have to be made. The decision about any controversial subsets could be made by any of the methods mentioned. An average, a minimum, or a maximum could be computed and utilized by such collaboration methods.

- *Incremental voting.* An agent is selected by some means already discussed. One agent's result is compared to another's and, if they are the same, the result is forwarded. If the comparison is different or if more comparisons are desired, then more agents are included before a result is forwarded. A variation to this would be for agents to sample a subset of the data and compare results. Agents who differ from the majority are culled from the sampling, and comparisons continue until a single result emerges.

A combination of the preprocessing and postprocessing approaches could also be used. For example, more than one agent could be selected, using either a random or a voting preprocessing approach and the result would then be selected using one of the postprocessing approaches.

The preprocessing approaches by themselves are not representative of our redundancy-based robustness proposal; however they are useful when implementing a combined approach.

For the application examples presented in the next section, we controlled all of the activities (collection of the Web services, development of the agents and integration). In practice, there are different modes in which our approach can be implemented:

- *An intermediary approach.* A third-party business accesses the Web services, provides agents for different combining approaches, and exposes the multiagent system as a single Web service that will provide the result produced by agreement among the agents.

- *WS consumers combine the redundant functionality.* In order for this option to be viable, agent implementations of the combining approaches would have to be available, as well as tools for integrating them with the Web services.

- *WS providers include the agent capabilities into their exposed Web services.* Standards for agent-based WS would need to emerge for the WS providers to be able to do that.

Finally, the redundancy provider (one of the three above) would have to decide on the number of WS and thus, the number of agents. More agents lead to more robustness, but communication overhead and processor time are limiting factors. Hence the redundancy provider needs to manage the trade-off between cost and required reliability.

5. APPLICATION EXAMPLES

We collected a number of Web services, each offered by a different provider. We then created an agent for each Web service. The agent provides access to the Web service (through its WSDL definition) and has incorporated capabilities to interact with others in order to jointly agree on a solution. The agents were written in JADE[9] and make use of the Java JAX-RPC[10] and SAAJ[11] APIs, as well as the Apache AXIS SOAP implementation[12]. The agents can be viewed as SOAP clients with agent capabilities: i.e., protocols for handling communication, negotiation, and interaction. We worked with WS for two different domains: weather information and data sorting. In the following subsections, we explain how we achieved robust Web service functionality for each of these domains.

[9] http://sharon.cselt.it/projects/jade/
[10] http://java.sun.com/xml/jaxrpc/index.jsp
[11] http://java.sun.com/xml/saaj/index.jsp
[12] http://ws.apache.org/axis/

5.1 WEB SERVICES FOR DATA SORTING

The sorting problem is defined as follows (5):

- Input: A sequence of n numbers $(a_1, a_2, ..., a_n)$
- Output: A permutation (reordering) $(a'_1, a'_2, ..., a'_n)$ of the input sequence, such that $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Sorting is a fundamental operation in computer science (many programs use it as an intermediate step), and as a result a large number of good sorting algorithms have been developed.

Table 14.2. Sorting Algorithms Collected

Algorithm	*Features*
C.A.R Hoare's Quick Sort	Input (java) data type: int array Positive and negative numbers accepted
HeapSort	Input (java) data type: int array Positive and negative numbers accepted
QuickSort	Input (java) data type: Byte array, Short array, Integer array, Long array, Float array, Double array, String array, Character array Only positive numbers accepted
RadixSort	Input (java) data type: int array Only ten inputs accepted

For the development of this application we performed the following steps:

- We collected a number of sorting algorithms, each provided by a different programmer and therefore having different input and output signatures and performance characteristics. Table 2 summarizes the algorithms collected and information about them.
- We made each algorithm available as a WS. Unlike the Weather Information case described below, for this application we created the Web services ourselves and publicized them (instead of using available services). Available tools make it easy to perform this task. We used AXIS Java2WSDL[13] for generating the WSDL specification files from the source code files of the sorting algorithms, AXIS WSDL2Java[14] for

[13] http://ws.apache.org/axis/java/ant/axis-java2wsdl.html
[14] http://ws.apache.org/axis/java/ant/axis-wsdl2java.html

generating the sorting Web services' interfaces, and the skeleton of the classes from the WSDL specification files. We then filled the skeleton files manually using the source code files of the sorting algorithms. To make the sorting Web services available we used Tomcat[15].

- We then created a client for each Web service. AXIS WSDL2Java can also be used to create files for the clients' interfaces and classes: it automatically generates the Java source code necessary to access the Web service described in the WSDL file that it gets as input. We used it for that purpose and then compiled the generated files.

- We then used JADE for writing an agent for each client. JADE agents communicate using the FIPA Agent Communication Language[16] messages. Table 3 shows the core code of each agent. JADE automatically invokes the setup method after the creation of the agent. On setup, each agent performs the following steps

  - Register SL (FIPA Semantic Language[17]) as the content language to use for the ACL messages.
  - Register the ontology that they will use in those messages, which defines the terms that the agent will be able to understand. We defined an ontology called SortingOntology using JADE support for ontologies[18]. Our ontology consist of three elements: the RequestAgentAction agent action, the ResultPredicate predicate, and the ErrorPredicate predicate. Table 4 lists the vocabulary that each one defines.
  - Register its service with the Directory Facilitator agent (DF).
  - Initiate all the behaviors that the agent can handle. We defined a behavior for each of the approaches for combining the agents' functionalities that the agent is able to involve. We implemented a combination of a distributed preprocessing approach and a performance-based postprocessing approach (the agent whose processing was the fastest), as well as a voting postprocessing approach (the result given by most agents)[19]. Figure 2 depicts a UML diagram of the logistics of the system for the combined approach. The agents use the FIPA-request protocol to communicate and agree on a solution. Upon a user request, a BrokerAgent performs a search with the DF

[15] http://jakarta.apache.org/tomcat/
[16] http://www.fipa.org/specs/fipa00061/
[17] http://www.fipa.org/specs/fipa00008/
[18] sharon.cselt.it/projects/jade/doc/CLOntoSupport.pdf
[19] The application can be accessed at http://www.cse.sc.edu/ zavalagu/redundancy/sorting

for all the sorting agents available and sends a Request message with a RequestAgentAction as content to all of them. Each agent decides, based on the input data type, whether its component is capable of executing the user request. Any agent whose component is capable of executing the user request attempts to run it and sends to the BrokerAgent either an Inform message with a ResultPredicate as content, or a Failure message with an ErrorPredicate as content. The first valid result received by the BrokerAgent is drawn. In this way, over a period of time, the fastest and most available components are chosen the most often, although some connections are very slow or sometimes not available.

Table 14.3. Agents' Setup Method

```
protected void setup()
{
    manager.registerLanguage(new jade.content.lang.sl.SLCodec());
    manager.registerOntology(SortingOntology.OntologyScheme.getInstance());
    registerService();
    addBehaviour(new PerformanceBasedBehaviour(this));
    addBehaviour(new VotingBehaviour(this));
}
```

Taken together, our agents are applicable to a wider set of scenarios than any individual sorting Web service. Some of the agents accept only integer numbers as input, others can handle floating-point numbers, and others accept strings. One allows ascending and descending ordering, and the rest only ascending. Finally, some connections are faster and steadier than others.

Table 14.4. Terms defined in the SortingOntology ontology

Element	*Vocabulary*
RequestAgentAction	jade.util.leap.List elements java.lang.String elementsType
ResultPredicate	jade.util.leap.List result
ErrorPredicate	java.lang.String errorMessage

5.2 WEB SERVICES FOR WEATHER INFORMATION

For this case, we made use of available Web capabilities provided by different parties. Many repositories of Web services are publicly available for use and

some of them include weather report capabilities. The basic functionality that each service provides consists of giving the temperature for some locale in the world. Some provide additional functionalities, such as forecasts. Table 5 lists the Web services that we used and their providers.

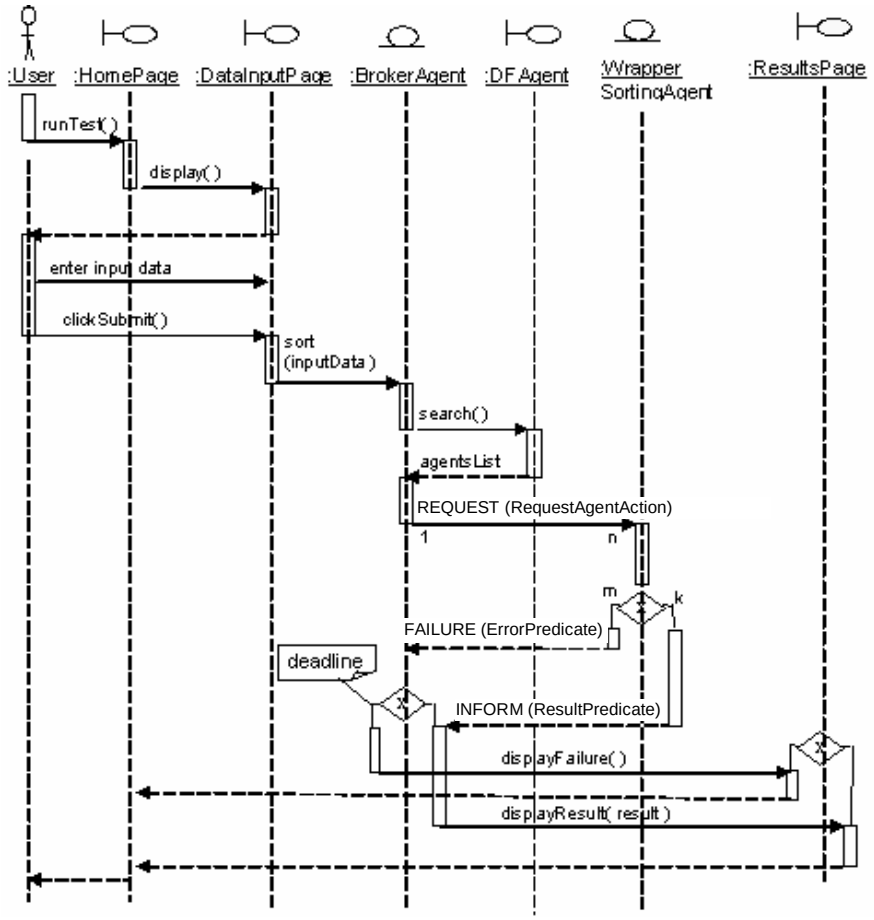

Figure 14.2. Interaction logistics of the agent-based Web services for sorting

We generated the client for the weather WS in exactly the same way that we did for the sorting WS. The creation of the agents for the weather WS was also the same as for the sorting WS, except that for this experiment we only implemented one approach for combining the agents' functionalities and we defined a new ontology called WeatherOntology.

We implemented a combination of a distributed preprocessing approach and a performance-based postprocessing approach[20]. The protocol that the agents use is the same as the one used for the Sorting domain and is depicted in Figure 2.

Again, taken together, our weather agents work for a wider set of scenarios than any individual weather Web service. Some of them accept only a USA 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 USA (asking for the zip code), while another can give the weather for almost any country by means of an airport code. Some of them give temperatures in Fahrenheit, others in Celsius, and others in both scales. Also, some WS provide only the current temperature, while others provide additional information such as n-day forecasts, humidity, sky, wind, visibility, location, etc. The WS vary also on the additional methods provided to make their use easier, such as a list of the countries for which they can give the weather, a list of airports in a particular country or region, a list of regions in a country, etc. Finally, some connections are faster than others as well as having more constant throughput.

6. FUTURE WORK

Just as there are different ways that a group of people can reach conclusions and make decisions, so are there different ways that a group of agent-wrapped software components can combine their results. We are continuing to study alternative ways for combining the agents' functionality; i.e., a combination of preprocessing and postprocessing 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 postprocessing approach, we could use the average of the results (for the particular domain, such as weather).

Our interest is in experimentation on large-scale systems; i.e., wrapping agents around redundant software components written in different computer languages, using different operating systems, 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

[20]The application can be accessed at http://www.cse.sc.edu/ zavalagu/redundancy/testbed

Table 14.5. Weather Web Services and its Providers

Provider	*URL*	*Web Service Functionality*
InnerGears Web Services (http://www.innergears.com/ WebServices.aspx)	http://www.innergears.com/ WebServices/WeatherByZip/ WeatherByZip.asmx?WSDL	Temperature by zip code
InnerGears Web Services (http://www.innergears.com/ WebServices.aspx)	http://www.innergears.com/ WebServices/WorldWeatherByICAO/ WorldWeatherByICAO.asmx?WSDL	Weather report by airport code
InnerGears Web Services (http://www.innergears.com/ WebServices.aspx)	http://www.innergears.com/ WebServices/StateWarnings/ StateWarnings.asmx?WSDL	Warnings report by state
InnerGears Web Services (http://www.innergears.com/ WebServices.aspx)	http://www.innergears.com/ WebServices/ForecastByZip/ ForecastByZip.asmx?WSDL	Forecast report by zip code
XMethods Demo Services (http://www.xmethods.net)	http://www.xmethods.net/sd/ TemperatureService.wsdl	Temperature by zip code
Capescience (capescience .capeclear.com/ Webservices /index.shtml)	http://live.capescience.com/wsdl/GlobalWeather.wsdl	Weather report by country, region, search keyword, and airport code
Capescience (capescience .capeclear.com/ Webservices /index.shtml)	http://live.capescience.com/ wsdl/AirportWeather.wsdl	Humidity, location, pressure, sky conditions, summary, temperature, visibility, and wind by airport code
Juice Software (Webservices .juice.com)	http://Webservices.juice.com :4646/temperature.wsdl	Temperature by zip code
UNISYS Sample Web Services (http://www.unisysfsp .com/ default.aspx?catID=17)	http://weather.unisysfsp.com/ PDCWebService/ WeatherServices .asmx?WSDL	Temperature and weather report by zip code
Ejse Web Services	http://www.ejse.com/ WeatherService/ Service.asmx?wsdl	Weather report by zip code and Iraq city name

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.

ACKNOWLEDGMENTS

This work was supported in part by the Advanced Research and Development Activity (ARDA). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Government. This work was also supported by the U.S. National Science Foundation under grant number IIS-0083362.

REFERENCES

1. Algirdas Avizienis, "The Methodology of N-version Programming," Software Fault Tolerance, edited by M. Lyu, John Wiley & Sons, 1995, pp. 23–46.

2. Paul Buhler and Jose M. Vidal, "Semantic Web services as Agent Behaviors," Agentcities: Challenges in Open Agent Environments, edited by B. Burg, J. Dale, T. Finin, H. Nakashima, L. Padgham, C. Sierra, and S. Willmott, Springer-Verlag, Berlin, 2003, pp. 25–31.

3. Adam Cheyer, and David Martin, "The Open Agent Architecture," Journal of Autonomous Agents and Multi-Agent Systems, vol. 4 , no. 1, March 2001, pp. 143–148.

4. Helder Coelho, Luis Antunes, and L. Moniz, "On Agent Design Rationale," Proc. XI Simposio Brasileiro de Inteligencia Artificial, Fortaleza (Brasil), October 17–21, 1994, pp. 43-58.

5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein "Introduction to Algorithms," The MIT Press, Cambridge, Massachusetts, London, England, McGraw-Hill Book Company, Second Edition, 2001.

6. Jonathan Dale, Akos Hajnal, Martin Kernland, and Laszlo Zsolt Varga, "Integrating Web services into Agentcities", Agentcities Technical Recommendation Document. Available at: http://www.agentcities.org/rec/00006/

7. Vance T. Holderfield and Michael N. Huhns, "A Foundational Analysis of Software Robustness Using Redundant Agent Collaboration," Proc. Int'l Workshop on Agent Technology and Software Engineering, Erfurt, Germany, October 2002.

8. Michael N. Huhns, Vance T. Holderfield, and Rosa Laura Zavala Gutierrez, "Achieving Software Robustness Via Large-Scale Multiagent Systems," Software Engineering for Large-Scale Multi-Agent Systems, edited by A. Garcia, C. Lucena, F. Zambonelli, A. Omicini, and J. Castro, Springer Verlag, Lecture Notes in Computer Science, Volume 2603, Berlin, 2003, pp. 199–215.

9. Michael N. Huhns: "Software Agents: The Future of Web Services," Agent Technologies, Infrastructures, Tools, and Applications for E-Services: NODe 2002 Agent-Related Workshops, Erfurt, Germany, October 7–10, 2002, edited by R. Kowalczyk, J. P. Müller, H. Tianfield, R. Unland, Springer-Verlag Heidelberg, Lecture Notes in Computer Science, 2592, 2003, pp. 1–18.

10. Michael N. Huhns, "Agents as Web services," IEEE Internet Computing, Volume 6, 2002, pp. 93–95.

11. Nicholas R. Jennings, "On Agent-Based Software Engineering," Artificial Intelligence 117, 2 (2000), 277–296.

12. David Martin, Adam Cheyer and Douglas Moran, "The Open Agent Architecture: A Framework for Building Distributed Software Systems," Applied Artificial Intelligence, vol. 13, no. 1-2, 1999, pp. 91–128.

13. Brian Randell, "System Structure for Software Fault-Tolerance," IEEE Transactions on Software Engineering, Vol. SE-1, pp. 220-232, 1975.

14. Brian Randell and Jie Xu, "The Evolution of the Recovery Block Concept," Software Fault Tolerance, edited by M. Lyu (Trends in Software series), pp.1–22, John Wiley & Sons, 1995.

15. Reid G. Smith, "The contract net protocol: High-level communication and control in a distributed problem solver," Readings in Distributed Artificial Intelligence, edited by A. H. Bond and L. Gasser, Morgan Kaufmann Publishers Inc., California, 1988, pages 357–366.

16. Ian Sommerville, "Software Engineering," Fifth edition, Addison-Wesley, 1995