Robust Software Via Agent-Based Redundancy

Michael N. Huhns
University of South Carolina
Dept. of Computer Science & Engr.
Columbia, SC 29208 USA
+1-803-777-5921
huhns@sc.edu

Vance T. Holderfield
University of South Carolina
Dept. of Computer Science & Engr.
Columbia, SC 29208 USA
+1-803-777-xxxx
vance@sc.edu

Rosa Laura Zavala Gutierrez
University of South Carolina
Dept. of Computer Science & Engr.
Columbia, SC 29208 USA
+1-803-777-xxxx
zavalagu@engr.sc.edu

ABSTRACT

This paper describes how multiagent systems can be used to achieve robust software, one of the major goals of software engineering. The paper first positions itself within the software engineering domain. It then develops the hypothesis that robust software can be achieved through redundancy, where the redundancy is achieved by agents that have different algorithms but similar responsibilities. The agents are produced by wrapping conventional algorithms with a minimal set of agent capabilities, which we specify. We show that the same wrapper can be used for a variety of algorithms. We describe our initial experiments in verifying our hypothesis and present results that show an improvement in robustness due to redundancy. We also completely characterize the decision-making that must occur to decide among competing redundant algorithms. We conclude by speculating on the implications of multiagent-based redundancy for general software development and future Web services.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence – *multiagent systems.*

General Terms

Reliability, Experimentation, Theory

Keywords

Robust software; autonomic computing

1. INTRODUCTION

When discussing the uses and applications of agents with software developers from business, industry, and academia over the years, we have often heard the claim, "Agents might be appropriate for building a distributed X, but clearly you wouldn't use agents to build a sorting algorithm." The purpose of this paper is to show that if the requirements include either robust behavior or next-generation Web services, then agents are the *right* technology for constructing algorithms such as sorting.

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 typically are complex, involve massive amounts of data, affect numerous connected devices, and are subject to the uncertainties of open environments like the Internet. Our society has come to expect uninterrupted service from these systems. Unfortunately, when problems arise, humans are unable to cope with the complexity of the systems and the speed with which they must be repaired. Increasingly, the result is that critical missions are in jeopardy.

To cope with this situation, companies and researchers are investigating self-monitoring and self-healing systems, which detect problems autonomously and continue operating by fixing or bypassing the malfunction [12]. The techniques employed include redundant hardware, error-correction codes, and, most importantly, models of how a system *should* behave, so that the system can recognize when it *mis*behaves.

Such techniques can work well for hardware, but not for software, where having identical copies of a module provides no benefit. Software reliability is thus an unresolved problem. The amount of money lost due just to software errors is conservatively estimated at US$40B annually. One alternative is to produce higher quality software. To achieve this, researchers are trying to define more principled methodologies for software engineering [2]. They are also looking to new technologies, such as multiagent systems, and this leads to a natural interest in combining the latest software engineering methodologies with multiagent systems [10].

There are three ways that software engineering intersects multiagent systems:

1. Multiagent systems can be used to aid traditional software engineering; e.g., in the form of agent-based or agent-supplemented CASE tools
2. Traditional or new software engineering techniques can be used to build multiagent systems; e.g., the Unified Modeling Language (UML) has proven to be useful for conventional software, so Agent-Based UML (AUML) and similar efforts are underway to extend UML to support agent development
3. Conventional software can be constructed out of agents, and software engineering techniques can be developed to aid in this endeavor.

The focus of this paper is on the last.

Conference AAMAS'03, July 1-2, 2003, Melbourne, Australia.

2. BACKGROUND

Software engineering principles applied to multiagent systems have yielded few new modeling techniques, despite many notable efforts. A comprehensive review of agent-oriented methodologies is contained in Iglesias, et al. [6]. Many, such as Agent UML [11] and MAS-CommonKADS, are extensions of previous software engineering design processes. Others, such as Gaia [14], were developed specifically for multiagent system design. It was formulated from an organization theory perspective, with a methodology based on a model consisting of roles, permissions, responsibilities, protocols, activities, liveness properties, and safety properties. Because agents are useful in such a broad range of applications, software engineering methodologies for multiagent systems should be a combination of principles and techniques. This combination of will generally give a more flexible approach to fit a design team's particular expectations and requirements.

Multiagent systems can form the fundamental building blocks for software systems, even if the software systems do not themselves require any agent-like behaviors [7]. When a conventional software system is constructed with agents as its modules, it can exhibit the following benefits [1][4]:

1. Agents enable dynamic composibility, where the components of a system can be unknown until runtime
2. Agents allow interaction abstractions, where interactions can be unknown until runtime
3. Because agents can be added to a system one-at-a-time, software can continue to be customized over its lifetime, even potentially by end-users
4. 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.

2.1 Bugs, Errors, and Redundancy

Hardware robustness is typically characterized in terms of faults and failures; equivalently, software robustness is typically characterized in terms of bugs and errors. Faults and bugs are flaws in a system, whereas errors and failures are the consequences of encountering the flaws during the operation or execution of the system. The flaws may be either transient or omnipresent. The general approaches for dealing with flaws are the same for both hardware and software: (1) prediction and estimation, (2) prevention, (3) discovery, (4) repair, and (5) tolerance or exploitation.

Fault and bug estimation uses statistical techniques to predict how many flaws might be in a system and how severe their effects might be. For example, when Windows XP was released, it was estimated that it still contained 60,000 bugs, based on the rate at which its bugs were being discovered. Bug prevention is dependent on good software engineering techniques and processes. Good development and run-time tools can aid in bug discovery, whereas repair and tolerance depend on redundancy.

Indeed, redundancy is the basis for most forms of robustness. It can be provided by replication of hardware, software, and information, and by repetition of communication messages. For years, NASA has made its satellites more robust by duplicating critical subsystems: if a hardware subsystem fails, there is an identical replacement ready to begin operating. The space shuttle has quadruple redundancy, and won't leave the ground without all copies functioning. However, software redundancy has to be provided in a different way. Identical software subsystems will fail in identical ways, so extra copies do not provide any benefit.

Moreover, code cannot be added arbitrarily to a software system, just as steel cannot be added arbitrarily to a bridge. When we make a bridge stronger, we do it by adding beams that are not identical to ones already there, but that have equivalent functionality. This turns out to be the basis for robustness in software systems as well: there must be software components with equivalent functionality, so that if one fails to perform properly, another can provide what is needed. The challenge is to design the software system so that it can accommodate the additional components and take advantage of their redundant functionality.

We hypothesize that agents are a convenient level of granularity at which to add redundancy and that 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 [4]. Agents by design know how to deal with other agents, so they can accommodate additional or alternative agents naturally. They are also designed to reconcile different viewpoints.

Fundamentally, the amount of redundancy required is well specified by information theory. Assume each software module in a system can behave either correctly or incorrectly. Then two modules with the same intended functionality are sufficient to detect an error in one of them, and three modules are sufficient to correct the incorrect behavior (by voting, or choosing the best two-out-of-three). This is exactly how parity bits work in code words. Unlike parity bits, and unlike bricks and steel bridge beams, however, the software modules cannot be identical, or else they would not be able to correct each other's errors.

If we want a system to provide n functionalities robustly, we must introduce $m \times n$ agents, so that there will be m ways of producing each functionality. Each group of m agents must understand how to detect and correct inconsistencies in each other's behavior, without a fixed leader or centralized controller. If we consider an agent's behavior to be either correct or incorrect (binary), then, based on a notion of Hamming distance for error-correcting codes, $4 \times m$ agents can detect $m-1$ errors in their behavior and can correct $(m-1)/2$ errors.

Redundancy must also be balanced with complexity, which is determined by the number and size of the components chosen for building a system. That is, adding more components increases redundancy, but also increases the complexity of the system. This is just another form of the common software engineering problem of choosing the proper size of the modules used to implement a system. Smaller modules are simpler, but their interactions are more complicated because there are more modules.

An agent-based system can cope with a growing application domain by increasing the number of agents, each agent's capability, the computational resources available to each agent, or the infrastructure services needed by the agents to make them more productive. That is, either the agents or their interactions

can be enhanced, but to maintain the same degree of redundancy *n*, they would have to be enhanced by a factor of *n*.

To underscore the importance being given to redundancy and robustness, several initiatives are underway around the world to investigate them. IBM has a major initiative to develop autonomic computing—"a systemic view of computing modeled after the self-regulating autonomic nervous system." Systems that can run themselves incorporate many biological characteristics, such as self-healing (redundancy), adaptability to changing environments (reconfigurability), identity (awareness of their own resources), and immunity (automatic defense against viruses). An autonomic computing system will adhere to self-healing, not by "cellular regrowth," but by making use of redundant elements to act as replenishment parts. By taking advantage of redundant services located around the world, a better range of services can be provided for customers in business transactions.

Exemplifying extreme redundancy in hardware, HP Labs has built a massively parallel computer, the Teramac, with 220,000 known defects, but it still yields correct results. As long as there is sufficient communication bandwidth to find and use healthy resources, it can tolerate the defects. Allowing so many defects enables the computer to be built cheaply.

The National Science Foundation has launched the IRIS project to produce a robust, decentralized, and secure Internet infrastructure. The infrastructure will be developed using distributed hash table technology, which can prevent all the data in a network from becoming vulnerable if one server crashes. Rather than centralizing the data in a single server, each server contains a partial list of the data's storage location.

2.2 N-Version Programming

N-version programming, also called dissimilar software, is a technique for achieving robustness first considered in the 1970's. It consists of *N* separately developed implementations of the same functionality. Although it has been used to produce several robust systems, it has had limited applicability, because (1) *N* independent implementations have *N* times the cost, (2) *N* implementations based on the same flawed specification might still result in a flawed system, and (3) the resultant system might have *N* times the maintenance cost (e.g., each change to the specification will have to be made in all *N* implementations).

2.3 Checkpointing, Rollback, Compensation

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. Consistent transactions are achievable for databases because the types of processing done are very regular and limited.

Applying this idea to general software execution requires that the state of a software system be saved periodically (a checkpoint) so that the system can return to that state if an error occurs. The system then returns to that state and processes other transactions or alternative software modules. This is depicted in Figure 1.

There are two ways of returning to a previous state: (1) reloading a saved image of the system before the failed computation, or (2) rolling back, i.e., reversing and undoing, each step of the failed computation. Both of the ways suffer from major difficulties:

1. The state of a software system might be very large, necessitating the saving of very large images
2. Many operations cannot be undone, such as those with side-effects. Examples of these are sending a message, which cannot be un-sent, and spending resources, which cannot be un-spent. Rollback is successful in database systems, because most database operations do not have side-effects.

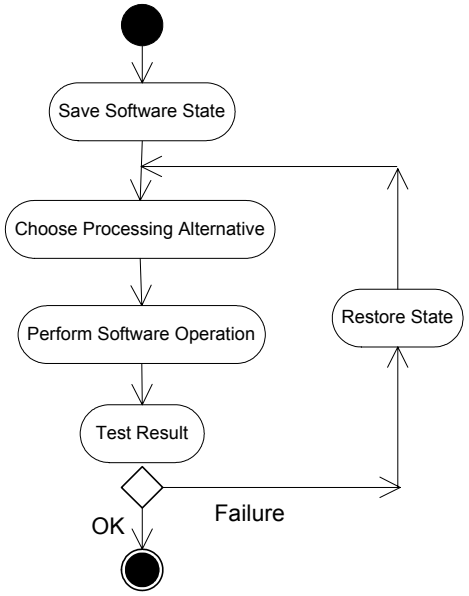

Figure 1. A transaction approach to correcting for the occurrence of errors in a software system

Because of this, compensation is often a better alternative for software systems. Figure 2 depicts the architecture of a robust software system that relies on compensation of failed operations.

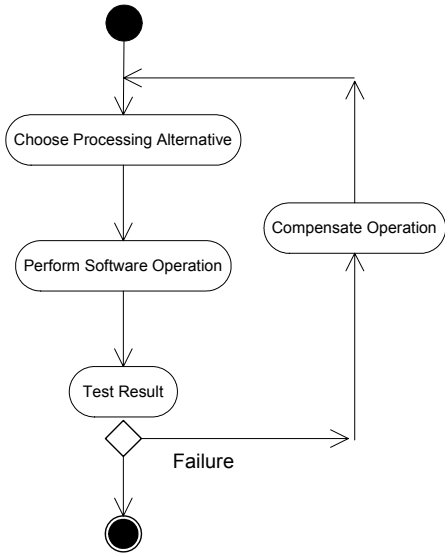

Figure 2. An architecture for software robustness based on compensating operations

3. ARCHITECTURE AND PROCESS

Suppose there are a number of sorting algorithms available. Each might have strengths, weaknesses, and possibly errors. One might work only for integers, while another might be slower but be able to sort strings as well as integers. How can the algorithms be combined so that the strengths of each are exploited and the weaknesses or flaws of each are compensated or covered? In solving this in a general way, we hypothesize that the end result is an "agentizing" of each algorithm.

A centralized approach, as shown in Figure 3, would use an omniscient preprocessing algorithm to receive the data to be sorted and would then choose the best algorithm to perform the sorting. Each module's characteristics have to be encoded into the central unit. The difficulties with this approach are (1) the preprocessing algorithm might be flawed and (2) it is difficult to maintain such a preprocessing algorithm as new algorithms are added and existing algorithms become unavailable.

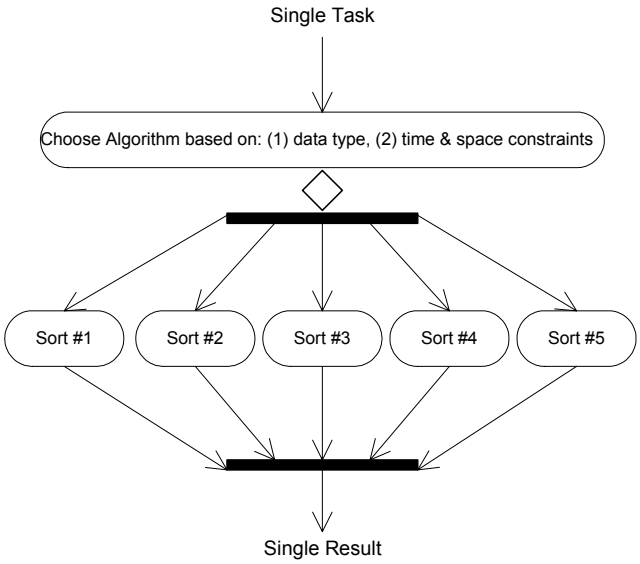

Figure 3. Centralized architecture for combining *N* versions of a sorting algorithm into a single, more robust system for sorting, where a preprocessing algorithm chooses which sorting algorithm will execute

An altermative is a postprocessing algorithm, as shown in Figure 4, that receives the results of all algorithms and chooses the best as the output. This approach is also centralized and suffers from a waste of CPU resources, because all algorithms work on the data and the results have to be compared. However, the comparison of outcomes is likely to produce better results.

A combination of the preprocessing and postprocessing centralized systems could also be used. Based on criteria known about each module, a subgroup could be selected to perform the desired task. The subgroup would then have its results compared to determine the best results as above.

A fourth approach is a distributed solution, where the algorithms jointly decide which one(s) should perform the sorting, and if there is more than one, they jointly decide on the best result. Conventional algorithms do not typically have such a distributed decision-making ability, so we investigate here whether there is a generic capability that can be added to an algorithm to enable it to participate in a distributed decision. The generic capability has the characteristics of a software agent. Distributing the centralized functions into the different modules creates a multiagent system.

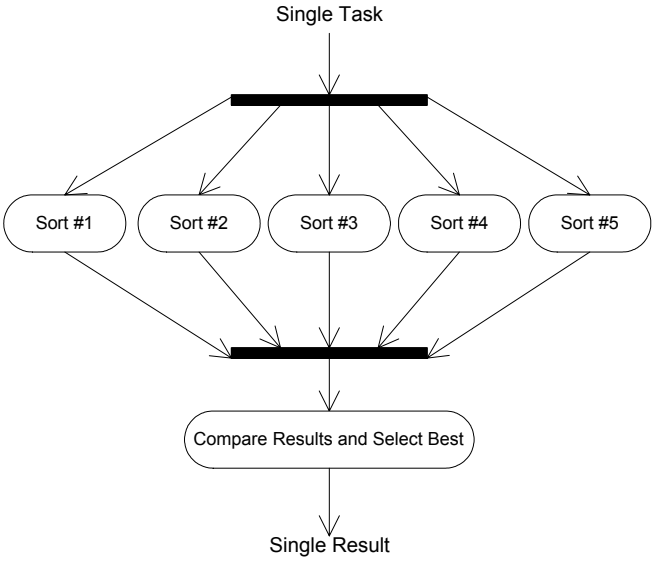

Figure 4. Centralized architecture for combining *N* versions of a sorting algorithm into a single, more robust system for sorting, where a postprocessing algorithm chooses one result to be the output

Each agent would have to know its role in this system as well as

- Something about its own algorithm, such as its time and space complexity, and input and output data structures
- Something about other agents, such as their time and space complexity and reliability
- How to negotiate
- How to communicate
- How to compare results
- How to manage reputations and trust.

An agent system derived from a redundant system with a centralized preprocessor might have the agents bid for a task. This would create an auction environment to determine task assignment. This is acceptable in a system with many competing agents, but the value of robustness based on reinforced redundant involvement is not achieved in an auction.

An agent system transformed from a redundant system with a centralized postprocessor would entail that each of the agents attempt the task and some type of voting mechanism (either with a voting factor or not) be used among the agents. A vote could be based on reputation or on a comparison of results. The communication overhead could be large.

The comparison of results can be done in several ways. The results can all be compared and a majority of exact outcomes would determine the results that are selected, where each of the agents have an equal chance at having their results selected. Alternatively, their chances could be weighted based on external

opinions of their past performances. Other factors could be based on information that the agent knows about itself, such as whether it completed the task or not, its time and space requirements, and the total number of runtime errors it has produced in the past.

4. EXPERIMENTS

We collected a number of algorithms, each of which belongs to one of two different types (i.e., sorting algorithms and list-reversing algorithms). Different people wrote each algorithm. For the sorting algorithms, no specifications were given to the programmers about how to write them, so the algorithms all have different characteristics, such as input data type, output data type, and time and space complexity. For the list-reversing algorithms, the structure of the class (i.e., methods names, results returned, types of parameters) was specified, so the differences among these algorithms are in performance and correctness. For our experiments, we converted each algorithm into a sorting agent.

Each agent is composed of an algorithm and a wrapper for that algorithm. The wrapper knows only about the external characteristics of its algorithm, such as its input and output data type(s) and its complexity, and nothing about the inner workings of the algorithm. The algorithms were written in JAVA and the wrappers in JADE.

The system sends data to be either sorted or reversed to all the corresponding agents (sorting agents or reversing agents, respectively). Their responsibility (as a group) is the sorting or reversing of the data, and they should be able to do this better than any one of them alone.

Two different approaches for combining the wrappers' functionality were implemented for the initial experiments: a distributed preprocessing approach and a centralized postprocessing approach. Each approach was tested with the sorting and reversing domains, one at a time. An architecture result we found is that the same wrapper can be used for both domains. An experimental result is verification of our hypothesis that more algorithms give better results than any one alone. However, it is important to note that the approach used for combining the wrappers' functionality plays an important role in obtaining that result.

4.1 Logic of the Preprocessing Approach

(For simplicity, we explain this approach with the sorting domain, because the results for the reversing domain are the same.) Upon receiving data to be sorted, each agent determines whether or not it can sort it successfully (based on the type of the data and its own knowledge of what types it can sort) and if the agent can, it broadcasts an INFORM message to every other agent specifying its intention, along with a measure of performance for its algorithm (based on execution time supplied by the agent itself).

The decision of which agent (i.e., algorithm) to choose among those that are capable of sorting the input data is made in a distributed manner: upon receiving the INFORM messages from other agents, each agent compares its expected performance against those received in the messages. If the agent has the best performance, it will run its algorithm and send the results back to the system. If it does not have the best, it will do nothing. Figure 5 depicts the AUML diagram of the protocol used. The system can be considered as the Initiator agent. Also, once they receive the data to be sorted from the system, the agents will wait for INFORM messages for only a limited amount of time; this avoids waiting infinitely long for messages from agents that either have problems sending a message or are not able to sort the data.

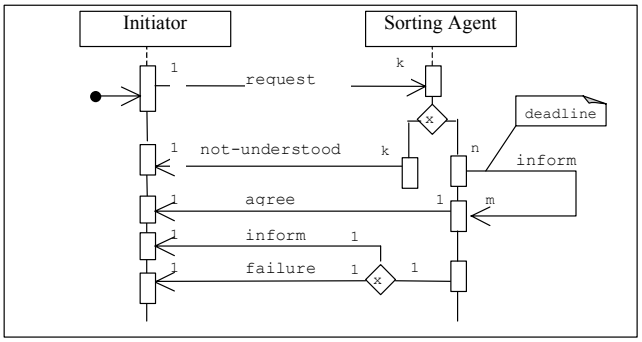

Figure 5. AUML diagram of the agents' interactions

4.2 Logic of the Postprocessing Approach

(We explain this, arbitrarily, using the reversing domain.) Upon receiving data to be reversed, *n* randomly chosen agents, where *n* is given by the user, will run their algorithms and send the results to the postprocessing system. The system compares the results and chooses the one that represents a majority of similar outcomes (one is selected randomly when there is not a clear-cut winner). Figure 6 depicts the AUML diagram of the protocol used. The system can be considered as the Initiator agent. Also, once the system sends to the wrappers the request message along with the data to be reversed and then receives the AGREE messages, it will wait for INFORM messages for only a limited amount of time; this avoids waiting infinitely for messages from agents that either have problems sending a message or reversing the data.

Table 1. Input data type and additional restrictions for the sorting algorithms available

Algorithm	Characteristics/Restrictions Used by Default Wrapper	Additional Restrictions
C.A.R Hoare's Quick Sort	Input data type: Integer array (positive and negative numbers accepted)	None
HeapSort	Input data type: Integer array (positive and negative numbers accepted)	None
QuickSort	Input data type: Byte array Short array Integer array Long array Float array Double array String array Char array (positive numbers only)	None
RadixSort	Input data type: int array	10 inputs

Table 1 summarizes the algorithms collected and information about them. The Additional Restrictions has restrictions not used by the default Wrapper to determine whether the agent can sort the data. Nevertheless, it is possible to cope with these cases by

providing an implementation of a WrapperRestrictions interface, which only has one method that takes as input the data to be sorted and returns a Boolean value indicating whether the sorting algorithm can sort the data. This provides a way of customizing the wrapper for algorithms that need additional considerations for the whole system to perform better.

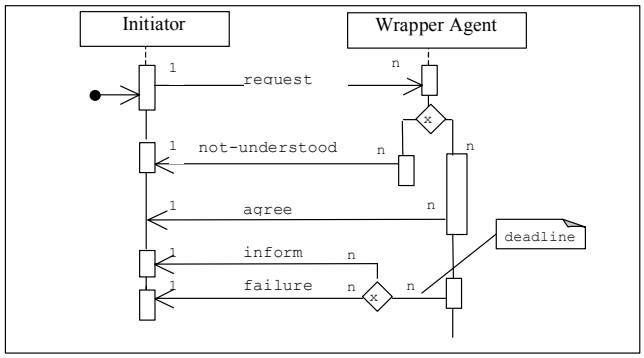

Fig. 6. AUML diagram of the agents' interactions

Our next set of experiments used 25 individually produced implementations of a doubly-linked list, each containing a method for reversing a list. The implementations differed in performance and correctness.

Table 2 provides a summary of the tests performed using the distributed preprocessing approach for the sorting domain. The best algorithm that can handle the input data is always chosen. It is different for the reversing domain, where the decision of which algorithm to execute is based on a measure of performance for each algorithm, which is calculated and stored by its agent. Unfortunately, a small value for time will be assigned for reversing algorithms that do nothing or that raise an exception (because they fail quickly). This will cause the wrong algorithms to be selected with the distributed preprocessing approach.

Table 2. Results for distributed preprocessing by agent-wrapped sorting algorithms

Data Input	Algorithm Selected	Data Output	Comments
12, 45, 3, 2, 56	C.A.R Hoare's Quick Sort	2, 3, 12, 45, 56	
ann, john, sue, marie	QuickSort	ann, john, marie, sue	Only this algorithm can handle strings
9, 8, 7, 4, 3, 2, 12, 4, 5, 10	RadixSort	2, 3, 4, 4, 5, 7, 8, 9, 10, 12	Best performance but only works for 10 inputs
3.54, 90, 23.4, 3.55, 60, 60.1	QuickSort	3.54, 3.55, 23.4, 60, 60.1, 90	Only this algorithm can handle reals

Finally, Table 3 provides a summary of the tests performed using the postprocessing approach in the reversing domain. This approach is generally better, but performs worse for cases with large data sets or selected algorithms with long execution times. An alternative is to base a decision on test results from each agent.

Table 3. Results for distributed postprocessing in the reversing domain

Data input	Selected Algorithm Results	Final Results
[9,5,3,1,8,7]	DList5 = [9,5,3,1,8,7] DList12 = [7,8,1,3,5,9] DList8 = [7,8,1,3,5,9]	[7,8,1,3,5,9]
[9,6,4,3]	DList16 = [3,4,6,9] DList11 = [3,4,6,9] DList23 = [3,4,6,9]	[3,4,6,9]
[3,5,2,9,11]	DList19 = [11,9,2,5,3] DList1 = Exception raised DList4 = [11,9,2,5,3]	[11,9,2,5,3]
[8,7,32,19]	DList24 = [19,32,7,8] DList9 = [19,32,7,8] DList14 = [19,32,7,8]	[19,32,7,8]

An agent-based wrapper, such as we have developed here for a redundant algorithm, can also serve as the basis for future agent-based Web services. Current practices for Web services suffer from the following limitations, which agents will be able to rectify:

- A Web service knows only about itself, but not about its users/clients/customers. Agents are often self-aware at a metalevel, and through learning and model building gain awareness of other agents and their capabilities as interactions among the agents occur. This is important, because without such awareness a Web service would be unable to take advantage of new capabilities in its environment, and could not customize its service to a client, such as by providing improved services to repeat customers.
- Web services, unlike agents, are not designed to use and reconcile ontologies. If the client and provider of the service happen to use different ontologies, then the result of invoking the Web service would be incomprehensible to the client.
- Agents are inherently communicative, whereas Web services are passive until invoked. Agents can provide alerts and updates when new information becomes available. Current standards and protocols make no provision for even *subscribing* to a service to receive periodic updates.
- A Web service, as currently defined and used, is not autonomous. Autonomy is a characteristic of agents, and it is also a characteristic of many envisioned Internet-based applications. Among agents, autonomy generally refers to social autonomy, where an agent is aware of its colleagues and is sociable, but nevertheless exercises its independence in certain circumstances. Autonomy is in natural tension with coordination or with the higher-level notion of a commitment. To be coordinated with other agents or to keep its commitments, an agent must relinquish some of its autonomy. However, an agent that is sociable and responsible can still be autonomous. It would attempt to coordinate with others where appropriate and to keep its commitments as much as possible, but it would exercise its

autonomy in entering into those commitments in the first place.

- Agents are cooperative, and by forming teams and coalitions can provide higher-level and more comprehensive services. Current standards for Web services do not provide for composing functionalities.

If Web services were wrapped with our agents, clients would be able to take advantage of redundant Web services, thereby attaining more reliable and robust results.

5. MULTIAGENT DECISION MAKING

The decision domain consists of identical agents wrapping dissimilar algorithms, which differ in time and space complexity, input and output data structures, and data type. Given such agents and a single task to be performed, a single result needs to be produced. The possible decision strategies in this domain can be classified into two basic categories, as summarized in Table 3. One category, preprocessing, involves choosing an agent before any data processing has begun, while the other category, postprocessing, would require some if not all the processing to be completed before deciding. By choosing one agent to deliver the output or the plan for specifying the output, preprocessing saves on CPU resources. By having many agents working redundantly on the same task, postprocessing uses more CPU resources. However, postprocessing decision strategies consider the actual output, while preprocessing decision strategies do not.

Table 3. Distributed decision strategies that are applicable to the agent wrapper system.

Preprocessing	Postprocessing
Random/lottery	Performance-based
Auction/election/criteria selection	Voting
Team	Collaboration
	Incremental

In a preprocessing decision strategy, there are three general methodologies that are applicable. The first and simplest would be to have one agent randomly chosen to perform the task, which 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.

Auctions, voting, and selection by a known set of criteria are all viable and similar options and are the second preprocessing methodology. Since this is a single input, single output subsystem, an agent's desire to perform the 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 the 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.

A team approach would be the third general methodology for a preprocessing decision strategy. This approach 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 could be divided into smaller subsets. If bugs or errors are present, however, this method will not find or correct them. A large team would require massive communication overhead. Teams also have to be organized and setup, which requires additional overhead.

In a *postprocessing* decision strategy where all agents process the data, there are four general methodologies that can be chosen. The first and simplest would be to have all the agents process the data and select the one whose processing was the fastest or used the least amount of space. The same problem would arise as with the preprocessing methods: not being able to determine or correct bugs and errors. A domain in which the agents are sufficiently competent would be an appropriate for this methodology.

A second postprocessing methodology would be to vote. 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. Alternatively, an evaluation mechanism could be applied by the other agents based on the following factors:

$$AgentEvaluation = f(\%Completed, \%Correct, Time\ used, Space\ used, Reputation)$$

where reputation is based on past successes in a group. Resource usage would increase due to the increased deliberations, so the methodology would be best with smaller groups of agents.

A collaboration methodology would require that data be compared between agents so that any common data subsets would be kept and only a decision about controversial data subsets would 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.

A final postprocessing methodology is an incremental one. 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.

The above decision-making approaches do not specify the number of agents. More agents lead to more robustness, but communication overhead and processor time are limiting factors. A messaging system where only a single message is sent from

agent-to-agent has *n!* complexity, where *n* is the number of agents. Large sets of data cause complexities for postprocessing decision strategies. Ultimately, the "best" algorithm is determined by

$$Performance \times Flexibility \times Reliability$$

where *Performance* is a function of time and space complexity, *Flexibility* is a function of how broad a range of input and output data structures the agent can handle, and *Reliability* is a measure of how well the agent can avoid run-time errors and exceptions.

6. CONCLUSION: CHALLENGES AND IMPLICATIONS FOR DEVELOPERS

Producing robust software has never been easy, and the approach recommended here would have major effects on the way that developers construct software systems:

- It is difficult enough to write one algorithm to solve a problem, let alone *n* algorithms. However, algorithms, in the form of agents, are easier to reuse than when coded conventionally and easier to add to an existing system, because agents are designed to interact with an arbitrary number of other agents.
- Agent organizational specifications need to be developed to take full advantage of redundancy.
- Agents will need to understand how to detect and correct inconsistencies in each other's behavior, without a fixed leader or centralized controller.
- There are problems when the agents either represent or use nonrenewable resources, such as CPU cycles, power, and bandwidth, because they will use it *n* times as fast.
- Although error-free code will always be important, developers will spend more time on algorithm development and less on debugging, because different algorithms will likely have errors in different places and can cover for each other.
- In some organizations, software development is competitive in that several people might write an algorithm to yield a given functionality, and the "best" algorithm will be selected. Under the approach suggested here, all algorithms would be selected.

Ultimately, the production of robust software will require that we understand the relationship between

- the social world as represented by humans and their physical environment, and
- the social world as represented by agents and other automated systems.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] Coelho, H., L. Antunes, and L. Moniz: "On Agent Design Rationale." In *Proc. XI Simposio Brasileiro de Inteligencia Artificial*, Fortaleza (Brasil), October 17-21, 1994, pp. 43-58.

[2] Cox, B. J.: Planning the Software Industrial Revolution. *IEEE Software*, (Nov. 1990) 25--33.

[3] Dignum, F., B. Dunin-Keplicz, and R. Verbrugge: "Dialogue in team formation: a formal approach" In van der Hoek, W., Meyer, J.J., and Wittenveen, C., Eds, *ESSLLI Workshop: Foundations and applications of collective agent based systems*, (1999).

[4] Holderfield, V.T. and M.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.

[5] Huhns, Michael N. and Vance T. Holderfield: "Robust Software," *IEEE Internet Computing*, vol. 6, no. 2, March-April 2002, pp. 80-82.

[6] C. Iglesias, M. Garijo, and J. Gonzalez: "A survey of agent-oriented methodologies." In J. Muller, M.P. Singh, and A.S. Rao, editors, *Proc. 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages* (ATAL-98). Springer-Verlag: Heidelberg, 1999.

[7] Jennings, Nick R.: "On Agent-Based Software Engineering" *Artificial Intelligence* 117, 2 (2000), 277-296.

[8] David Kalinsky: "Design Patterns for High Availability." *Embedded Systems Programming* (August 2002) 24--33.

[9] Laddaga, Robert: "Creating Robust Software through Self-Adaptation," *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999, pp. 26-29.

[10] Nwana, H.S. and Michael Wooldridge: "Software Agent Technologies." *BT Technology Journal*, 14(4):68-78 (1996).

[11] Odell, J., H. Van Dyke Parunak, and Bernhard Bauer: "Extending UML for Agents." In *Proceedings of the Agent-Oriented Information Systems Workshop*, Gerd Wagner, Yves Lesperance, and Eric Yu eds., Austin, TX, 2000.

[12] Paulson, Linda Dailey: "Computer System, Heal Thyself," *IEEE Computer*, (August 2002) 20--22.

[13] Schreiber, A. T., B. J. Wielinga, and J. M. A. W. Van de Velde: "CommonKADS: A comprehensive methodology for KBS development," 1994.

[14] Wooldridge, M., N. R. Jennings, and D. Kinny: "The Gaia Methodology for Agent-Oriented Analysis and Design." *J. Autonomous Agents and Multi-Agent Systems*, 2000.