Multiagent Reputation Management to Achieve Robust Software Using Redundancy

Rajesh Turlapati and Michael N. Huhns
Center for Information Technology, University of South Carolina
Columbia, SC 29208
{turlapat,huhns}@engr.sc.edu

Abstract

This paper explains the building of robust software using multiagent reputation. One of the major goals of software engineering is to achieve robust software. Our hypothesis is that robustness can be increased through redundancy. We achieve redundancy by using agents, with each agent wrapping a different algorithm with similar functionality. The agents build trust in each other using reinforcement learning. Two types of reputation management are simulated: one in which the reputations of all agents are maintained centrally and a second, which is distributed, where an agent maintains locally the reputations of the agents it knows and each agent can have its own evaluation of its known agents' performances. We simulated and compared two ways of achieving distributed reputation management. A probabilistic function is used as a preprocessing technique for selecting a set of agents based on reinforcement values of the agents. The values are obtained based on the correctness of the results the agent produces in performing the task it is given. Voting is used as a post-processing technique for judging the correctness of the output generated by the agents.

1. Introduction

Software redundancy has been shown to yield a high degree of fault-tolerance [3]. Software fault tolerance has become an implicit requirement in most application requirements. Any software system, from a simple home based one to a complex weather forecasting system is desired to be fault tolerant. Hardware redundancy is not sufficient, because any amount of redundant hardware can fail due to the same faulty software. Software redundancy is achieved by adding software components that are not exactly identical but have similar functionality. If one of the components fails, there might be an alternative component working in its place.

Many techniques have been developed for software redundancy, such as single-version software and multiple-version software. N-version programming (NVP) [2] is a process to build fault tolerant software systems. The major problems of NVP are how to maximize version development independence and minimize the probability of getting results that are identical when two or more versions are given the same task. *This paper extends N-version programming using multiagent redundancy, and solves some of the problems faced by the above technique.*

To enforce software redundancy, an agent can serve as a level of granularity and also an environment can be created to make the agents cooperate and learn among themselves. Multiagent systems learn not only by trial and error, but also through cooperation by sharing instantaneous information, episodic experience, and learned knowledge. Multiagent systems can form a basis for implementing redundancy and thereby provide a platform to achieve fault tolerant software. A system that has an agent-based infrastructure can increase its number of agents, the capability of each agent, and the resources available to these agents without introducing additional complexity. An earlier report [4] presents an idea that software redundant systems have non-identical software components that correct each others errors.

The agents are reinforced based on voting among the agents. Trust among agents is given special attention, because agents cannot always be presumed to operate in a cooperative environment free of fraud and deception. Formalization of trust [5] is very important in multiagent systems, and reputation helps in this formalization. An agent encounters problems of trust [8] when it is introduced into a competitive market.

Reinforcement learning is of great importance in multiagent systems, because the agents can learn about other agents through rewards obtained by performing tasks. Task success is decided by voting among the agents. Agents with correct results are given positive reinforcements and agents with incorrect results are given negative reinforcements. In our work, we use a probabilistic function as a preprocessing technique for choosing an initial set of agents. Agents with higher reinforcement values have a higher probability of being

selected than agents with lower reinforcement values. Voting is used as a post processing technique.

2. Problems of NVP

N-version programming suffers from the following problems:

- There is no control specified for choosing which algorithms to function.
- Simple voting is used, but nothing is specified to operate if the votes are tied.
- There is no learning in N-version programming, i.e., a faulty algorithm after being discovered is allowed to participate in system functioning.
- All of the versions do not communicate with one another or, in other words, they just execute the program and they have no knowledge about the other versions in the system
- Modifications of N-version programming can solve the problem of faulty algorithm executions through reputation management, but if they always execute algorithms with higher reputations, then versions with lower reputation will face a starvation problem
- Large amount of resources are consumed as all N-versions are executed in parallel.

3. Agent-based redundancy

The above problems with N-version programming can be mitigated by wrapping the components with agents. In order to be efficient, a mechanism is needed to choose which agents work on which problems, and it must function reliably even when the agents are uncooperative. We classify each agent into one of three categories based on the role it plays. An agent can assume any of the three roles depending on the situation.

Client agent:

A client agent is an agent with a task that it needs to have performed. Using central reputation management it can directly select service provider agents to perform the task or using distributed reputation management it can select a friend agent to perform the task.

Friend agent:

A friend agent comes into action in distributed reputation management. A friend agent recommends a service provider agent to the client agent.

Service provider agent:

A service provider agent wraps an algorithm, which is capable of performing some task.

In all forms of reputation management we simulated, execution starts with three distinct service providers selected using probabilistic agent selection and stops as soon as a consensus is reached. This method overcomes the problem of large resource consumption faced by NVP.

3.1. Probabilistic agent selection

This function selects an agent based on the reinforcement value. There is a bandwidth of values for an agent to be selected. This bandwidth is created in direct proportion with the reinforcement values of the agents. Maximum bandwidth is the sum of bandwidth of all the individual bandwidths. The random function generates a value with in this maximum bandwidth and corresponding agent is selected depending on the bandwidth to which this value belongs.

The probabilistic function is:

For each of client A's friends f_i, the probability $P(f_i)$ is

$$P(f_i) = \text{reputation}_i \,/\, \text{Sum of all reputations}$$

3.2. System configurations

3.2.1. Configuration 1. We collected algorithms from 25 students. Each algorithm implements a doubly linked list, with methods to insert and delete objects and traverse the list.

3.2.2. Configuration 2. We hypothesized a set of 25 algorithms, each with a different correctness percentage.

3.3. Central reputation management

In this scheme, the reputations of all agents are maintained centrally. A client agent knows about a service provider agent's reputation through this central reputation structure.

3.4. Experimental procedures

Each algorithm is treated as an agent implemented as a Java thread. There are a total of 150 inputs (50 Integer types, 50 Floating types, and 50 String types). Each input is 100 instances of Integers, Floats, or Strings. Positive reinforcement is an increment of 1 for every correct operation of the agent. Negative reinforcement is a decrement of 1 for every wrong operation of the agent. Reinforcements are awarded through post-processing techniques. I used voting among the agents. All agents are given same reinforcement values to start with. But it can be modified by giving different reinforcement values depending on various factors like time complexity that each algorithm takes. We used a preprocessing technique for selecting an initial set of 3 agents, selected randomly

based on the reinforcement values of the agents. Voting is done by comparing element values in a fixed position in all the lists. This fixed position is randomly generated.

3.5. Distributed reputation management

The correlation between trust and reputation is positive. So the first step to formalize trust is to formalize reputation. The formalization of reputation is restricted to a particular multiagent society because each society or an agent within a society has a different set of goals to be achieved. If an agent with the highest reputation transits to another group of multiagent systems it will have to re-establish its reputation. If no central authorities exist, the only way agents can find trustworthy agents is by exchanging information with others to identify those whose past behavior has been untrustworthy. Finding agents that are trustworthy reduces to the problem of distributed reputation management [1].

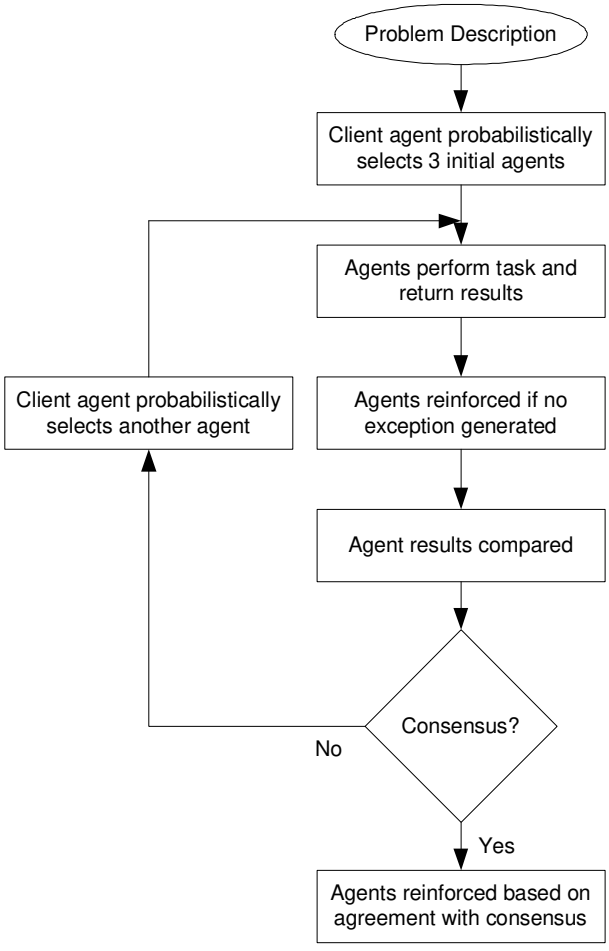

Figure 1. Central reputation management

Each agent maintains a local reputation management function. Whenever it needs service provider agents to perform a task it contacts the agents recommended in its local profiles. These agents are termed friend agents to distinguish them from service provider agents.

Distributed reputation management has several advantages over centralized management, as follows:

1) Each agent can have its own reinforcement method
2) Each agent can have different scales of evaluation
3) It is scalable simply by adding more agents, although an increase in redundancy might yield degraded performance depending on the application domain.

There are many ways of achieving distributed reputation management. We implemented two types of algorithms and compared the results.

3.6. Algorithm 1 assumptions and procedure

The client agent is chosen as agent number 1. Agent 1 starts with friend agent's list with an initial set of 5 agents. There are two setups used for configuration 2. In setup 1, the friends of all agents are chosen explicitly, but in setup 2, the friends of all agents are chosen randomly. For setup 1, the friends of agent 1 are 6, 7, 8, 9, and 10. Each of these friend agents starts with an initial reputation of 1. From its local profiles, the client agent selects three friend agents randomly, where the agents with better reputations are more likely to be chosen. **A service provider agent is selected randomly by each friend agent.** In case a service provider agent is already selected, another agent is selected randomly until 3 different service providers are selected.

If a service provider agent performs the task successfully, a positive reinforcement is given both to the agent selected and its referrer; if it fails, a negative reinforcement is given to the service provider and the referrer. If the agents disagree about their results and no result predominates, then the client agent contacts an additional friend agent through its probabilistic function. Positive reinforcement is incremented with 5 for every correct operation of the agent that is decided through voting. Negative reinforcement is decremented with 1 for every wrong operation of the agent.

3.7. Algorithm 2 assumptions and procedure

The client agent is chosen as agent number 1. Agent 1 starts with friend agent's list with an initial set of 5 agents

Two setups are used for configuration 2 just as in algorithm 1. For setup 1, agent 1's friends are 6, 7, 8, 9, and 10 and for setup 2 agent 1 chooses its friends at random. The client agent selects three friend agents from

its local profiles using probabilistic function. **Each friend agent selects its highest-ranked service provider agent.** In case a service provider is already selected, the agent with the next best ranking is selected until 3 different service providers are selected.

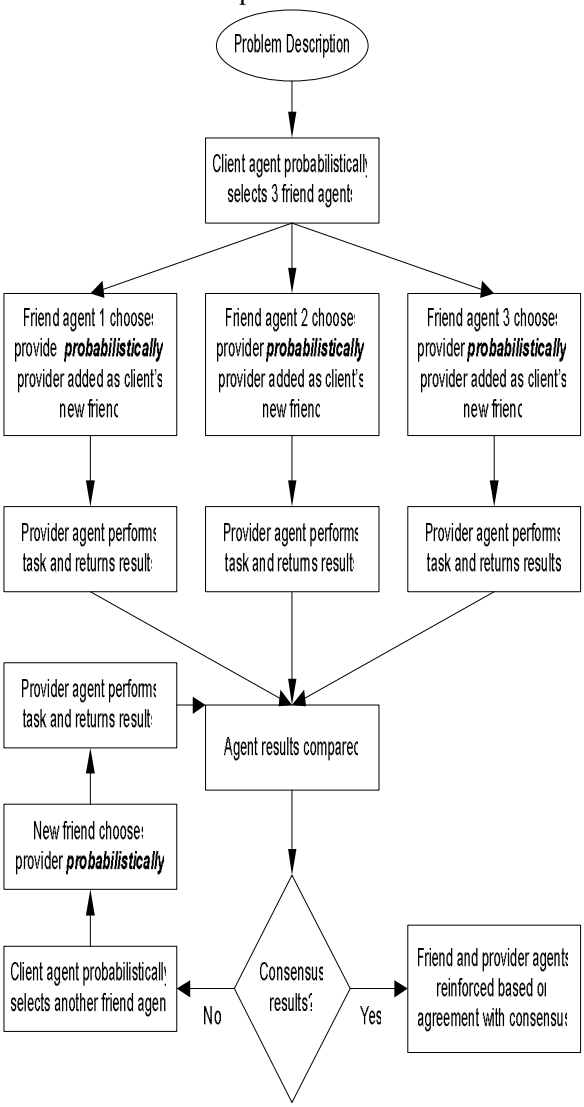

Figure 2. Distributed reputation management in algorithm 1

If an agent performs the task successfully, a positive reinforcement is given both to the agent selected and its referrer; if it fails, a negative reinforcement is given to the service provider and to the referrer. If the agents disagree about their results and no result predominates, then the client agent contacts an additional friend agent through its probabilistic function. The reputation is incremented by 5 for every correct operation of the agent and decremented by 1 for every wrong operation.

The main difference between algorithms 1 and 2 is in the way that service provider agents are selected. In the first algorithm, a probabilistic agent selection algorithm is used and in the second the agents with the best reputation are selected.

4. Results

4.1. Centralized reputation management configuration 1

Among 25 taken algorithms, 5 fail. But when all 25 algorithms are included in a robust software system, it achieves **100%** correctness. So the fault in one algorithm can be compensated for by the remaining algorithms when a single robust system is built.

4.2. Distributed reputation management

4.2.1. Algorithm 1, Configuration 1. (Table 1) The system is 100% correct Client agent 1 starts with 5 agents and increases its friend's list to a maximum limit of 24. There is an even distribution of reputations for all the agents

4.2.2. Algorithm 1, Configuration 2 (setups 1 and 2). (Table 2 and 3) The system is 96% to 100% correct. Client agent 1 starts with 5 agents and increases its friend's list to a maximum limit of 24. There is an even distribution of reputations for all the agents

4.2.3. Algorithm 2, Configuration 1. (Table 4) The system is 100% correct. The client agent 1 started with 5 agents and increased its friend's list to maximum limit of 24. There is a clustering of reputations for only some agents.

4.2.4. Algorithm 2, Configuration 2 (setups 1 and 2). (Table 5 and 6) The system is 96% to 100% correct. Client agent 1 starts with 5 agents and increases its friend's list to 18 after 100 service requests unlike in algorithm 1 where client agent 1 builds its friend's list with the maximum number of agents available in the environment. There is a clustering of reputations for only some agents.

4.3. Trade-offs between Algorithms 1 and 2

- Algorithm 1 builds its local reputation list in less time than algorithm 2.
- Algorithm 1 gives an even distribution of reputations among the local profiles of all the agents.

- Algorithm2 generates a consensus with fewer service provider agents per service request than algorithm 1.

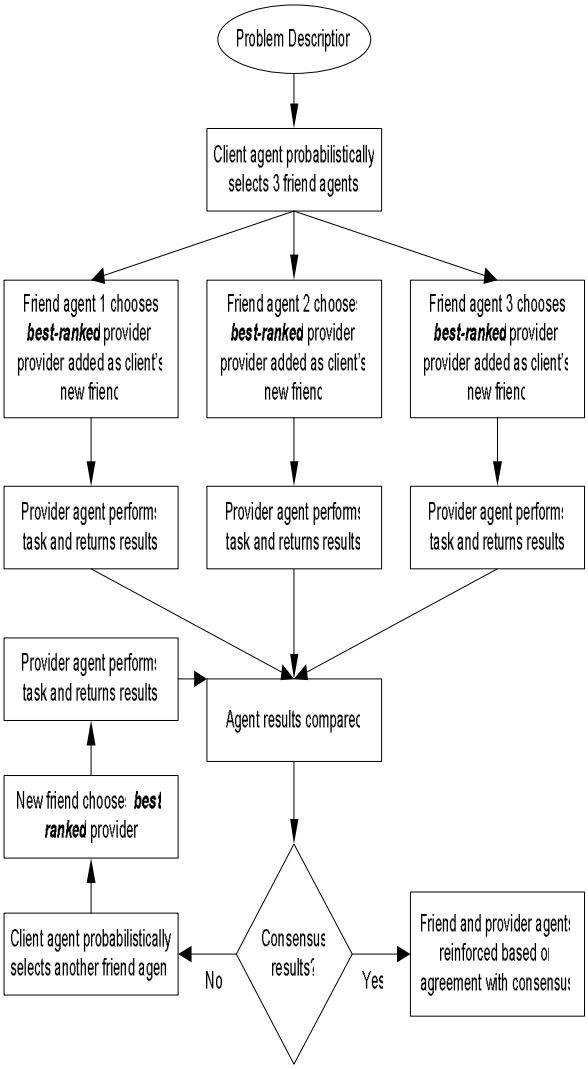

Figure 3. Distributed reputation management in algorithm 2

An attempt is awarded to a service provider agent if it tries to provide a service for a client and for a friend agent an attempt is awarded if it refers a service provider to a client. A correct is awarded to a service provider if it succeeds in performing a task as judged by voting and for a friend agent if the service provider referred by it succeeds in performing the task.

Correct Percentage = correct/attempted * 100

Votes are the units of reputation. Votes are awarded to an agent for performing its role correctly.

Table 1. Algorithm 1 – configuration 1 for client agent 1 with initial friends 6, 7, 8, 9 and 10. This shows the friends of client 1 and their reputations after 150 service (doubly linked list) requests

Friend	Votes
3	91
12	86
14	83
18	82
20	78
16	78
17	77
22	74
4	73
21	68
15	65
23	60
7	58
24	55
9	44
10	41
8	40
6	40
19	28
2	3
5	1
11	1
0	1

Table 2: Algorithm 1 – configuration 2 and setup 1 for client agent 1 with initial friends 6, 7, 8, 9 and 10. This shows the friends of client 1 and their reputations after 100 service requests.

Friends	Votes	Attempted	Correct %
12	41	20	33.33%
3	29	16	22.00%
20	27	14	27.91%
14	25	10	28.13%
22	25	5	41.18%
19	23	18	23.21%
8	21	14	20.45%
24	17	14	26.19%
16	17	17	33.33%
17	16	14	23.81%
18	15	8	16.67%

7	13	16	34.69%
13	13	17	25.49%
23	11	3	27.27%
4	8	7	26.09%
15	6	8	28.00%
11	6	14	25.00%
6	6	6	15.00%
5	3	5	11.76%
2	3	13	14.63%
9	1	7	18.18%
10	1	4	16.67%
0	1	8	0.00%
21	1	7	13.63%

Table 3: Algorithm 1 – configuration 2 and setup 2 for Client Agent 1 with initial friends 2, 7, 9, 4 and 13. This shows the friends of client 1 and their reputations after 100 service requests

Friends	Votes	Attempted	Correct %
21	45	77	33.77%
20	42	46	30.43%
13	36	62	24.19%
12	35	32	28.13%
22	33	43	32.56%
17	33	58	25.86%
9	33	63	30.16%
19	32	44	25.00%
16	30	61	26.23%
23	24	43	30.23%
15	20	45	28.88%
7	19	31	16.12%
11	16	19	26.32%
24	8	21	28.57%
10	6	23	17.39%
8	4	39	15.38%
2	1	14	14.29%
4	1	20	10.00%
5	1	8	0.00%
0	1	23	0.00%
14	1	9	0.00%
6	1	33	9.09%
3	1	33	6.06%
18	1	3	0.00%

Table 4: Algorithm 2 – configuration 1 for client agent 1 with initial friends 6, 7, 8, 9 and 10. This shows the friends of client 1 and their reputations after 150 service (doubly linked list) requests

Friends	Votes
13	183
18	182
23	145
14	141
19	115
24	100
12	79
22	63
7	59
20	47
17	43
4	31
8	26
16	25
3	23
9	21
21	18
6	9
11	7
15	6
0	4
10	1

Table 5: Algorithm 2 – configuration 2 and setup 1 for client agent 1 with initial friends 6, 7, 8, 9 and 10. This shows the friends of client 1 and their reputations after 100 service requests

Friends	Votes	Attempted	Correct %
22	115	56	41.42%
9	113	27	59.03%
23	19	19	18.97%
14	16	42	21.43%
8	11	10	15.63%
19	10	11	22.86%
17	7	20	11.48%
7	6	4	14.29%
12	5	3	11.11%
24	5	6	15.00%
2	5	5	29.41%
11	4	6	27.77%
6	1	1	0.00%
10	1	0	0.00%
18	1	4	15.39%
13	1	14	9.52%
0	1	4	8.33%

Table 6: Algorithm 2 – configuration 2 and setup 2 for client 1 with initial friends 16, 11, 24, 13 and

3. This shows the friends of client 1 and their reputations after 100 service requests

Friends	Votes	Attempted	Correct %
11	58	81	56.79%
24	46	143	37.76%
21	35	81	30.86%
16	32	50	34.00%
23	11	48	29.17%
9	6	76	14.47%
5	6	39	23.08%
22	3	65	23.08%
14	1	16	25.00%
13	1	6	0.00%
3	1	5	20.00%
0	1	7	0.00%
18	1	10	0.00%
4	1	18	5.56%
8	1	3	0.00%
19	1	3	0.00%
20	1	16	18.75%

4.4. Effect of redundancy

How much redundancy is needed depends on the application domain, for example:

4.4.1. An e-commerce system. There should be as much redundancy as possible. Increase in redundancy is not responsible for performance degradation as our system starts with 3 service provider agents and, if consensus is not achieved, then a fourth service provider agent is selected, and so on. Hence it does not matter if there are 100 or a billion agents in the society. The versions of software might have the same functionality but they are not identical. For example, agent 1 and agent 2 both sell goods. One might have expertise in selling books in which client A has an interest and one might have expertise in selling computers in which client B has an interest. In distributed reputation management, an agent can decide how many friend agents to keep track of. It might keep only 40 agents as friends, while another agent might keep 10,000 agents as friends. This is what we term as scalability in distributed reputation. All agents need not know about all agents. It is up to how an agent is implemented. Increased redundancy would affect a centralized reputation system as it maintains a global structure to store all reputations.

4.4.2. An operating system. Identify the critical points where a failure might occur. The number of versions of algorithms at the critical points depends on the budget, manpower, the risk at that point, etc.

5. Conclusion

We have shown that robustness is increased through redundancy, which is achieved through multiagent systems. The resulting system is robust and reliable. Selecting a centralized or distributed reputation management scheme depends upon the application domain.

The future work for this domain is extensive, and should be focused on

- What should be the number of algorithms?
- On what does the number of algorithms depend?
- What values of positive reinforcement and negative reinforcement are optimal?

6. References

[1] Bin Yu and Munindar P. Singh "An Evidential Model of Distributed Reputation Management," *AAMAS*, pp. 294-301, July 2002.

[2] V. Bharathi, "N-Version programming method of Software Fault Tolerance: A Critical Review," 721302, December 28-30, 2003.

[3] Wilfredo Torres-Pomales, "Software Fault Tolerance: A Tutorial", NASA/TM-2000-210616, pp. 66, October 2000.

[4] Michael N. Huhns, Vance T. Holderfield, and Rosa Laura Zavala Gutierrez, "Achieving Software Robustness via Large-Scale Multiagent Systems Vol. 2603, Berlin, pp. 199-215, 2003.

[5] Mladen A. Voukl, "An Empirical Evaluation of Consensus Voting and Consensus Recovery Block Reliability in the Presence of Failure Correlation*" *Journal of Computer and Software Engineering*, Vol. 1(4), pp. 367-388, 1993.

[6] Jonathan Carter, Elijah Bitting, and Ali A. Ghorbani, "Reputation formalization for an information Sharing multiagent system," vol. 18, no. 4, pp. 515–534, 2002.

[7] Leslie P. Kaelbling and Michael L. Littman, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research,* vol. 4, pp. 237-285, 1996.

[8] Sandip Sen and Neelima Sajja, "Selecting service providers based on reputation," AAAI Workshop 2002 WS-02-10 ISBN 1-57735-163-0.