Robust Software

Michael N. Huhns • *University of South Carolina* • *huhns@sc.edu*
Vance T. Holderfield • *University of South Carolina* • *vance@sc.edu*

Last week, we watched a neighbor build a brick wall next to his house. His old one had fallen down and he was determined not to let that happen again. "I'm going to make it twice as thick," he said. Earlier, we had asked a friend who was a bridge designer how he was sure the bridges he designed wouldn't collapse. He answered, "We calculate how much steel we will need to handle the expected stresses and strains, and then multiply by three."

As software developers, we would like the systems we construct to be robust and not crash. But we can't make them more robust simply by adding more code, as we add more bricks or steel to make a physical structure stronger. Otherwise Windows 98 with 20 million lines of code would be incredibly robust, and we haven't heard anyone make that claim!

Blindly adding code introduces more errors, makes the system more complex, and renders it harder to understand. However, adding more code can make software better, *if* it is added in the right way. As this article describes, the key concepts appear to be *redundancy* and the *appropriate granularity*.

Redundancy

Redundancy is the basis for most forms of robustness. For years, NASA has made its satellites more robust by duplicating critical subsystems. If a hardware subsystem fails, an identical replacement is ready to begin operating. The space shuttle has quadruple redundancy and won't leave the ground without all copies functioning. However, software redundancy must be provided in a different way. Identical software subsystems will fail in identical ways, so extra copies don't provide any benefit.

Moreover, we can't arbitrarily multiply the amount of code by three, just as steel can't 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: 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 capitalize on their redundant functionality.

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 multiple agents can fill each societal role. 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.

Theory of Redundancy

Fundamentally, information and coding theory has specified well the amount of redundancy required. 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 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 can't be identical, or else they could not 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,[1] m agents can detect $m - 1$ errors in their behavior and can correct $1/2(m - 1)$ errors.

Granularity

Fundamentally, system designers must balance

 1089-7801/02/$17.00 ©2002 IEEE

redundancy 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 might also increase the system's complexity. 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. 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.

Example Applications

Redundancy is a powerful tool to ensure a more robust result. Imagine an admissions committee considering an applicant to a university. The members of the admissions committee evaluate each applicant according to four criteria. They vote either "yes" or "no" depending on whether the applicant meets each criterion, and then overall on whether to grant admission. Now consider the following information in Table 1.

Table 1. Admission voting results.

Committee member	Good SATs?	Good grades?	Good letters?	Good writing samples?	Accept?
A	Yes	No	Yes	No	No
B	No	Yes	Yes	Yes	No
C	Yes	Yes	No	Yes	No

In this situation, the student would not be admitted to the university if the decision were left up to any individual committee member, even though the majority favored the student in each criterion. However, by aggregating the information in each criterion separately, the redundant composition of the committee can overturn the fallacies or biases of the individuals.[2]

Consider the more graphical example task show in Figure 1 of finding various routes from the University of South Carolina Visitor's Center to the Alumni House.

Individually, all three routes (red, yellow, and blue) are correct and achieve their goal of getting from the start to the finish. Due to redundancy, the system could achieve a consensus decision and obtain the more obvious, direct route (straight across). Moreover, the traversal execution is more robust, because it can overcome failures (blockages) in any of the paths.

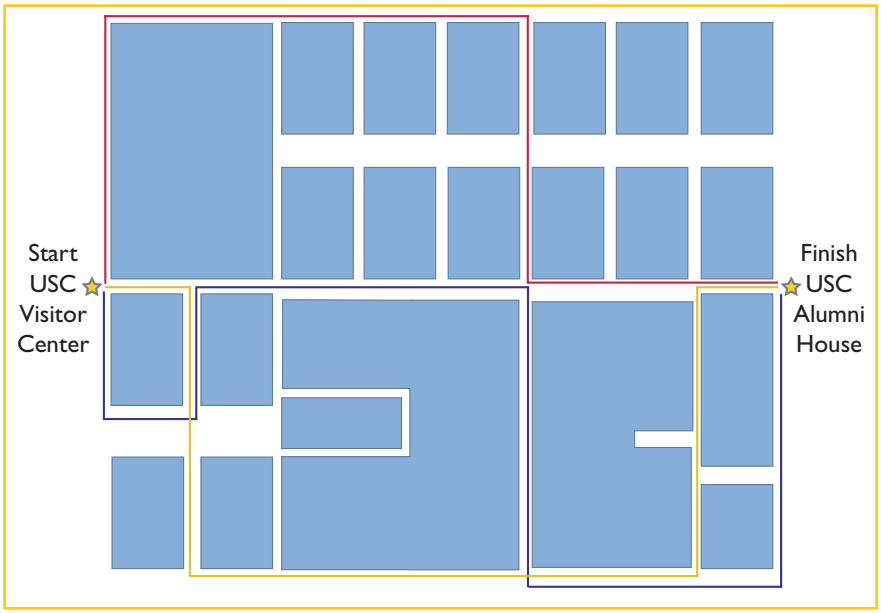

Figure 1. Three alternative (redundant) routes to a destination that could be combined into a single, more direct route. Because of the redundancy, blockage of any one route could be overcome, leading to more robust path traversal.

Successful teams in the RoboCup competition all make extensive use of redundancy. Team members each possess several different defensive and offensive strategies they can switch to at appropriate times. In each, players will support and back up their teammates. If a defender cannot win the ball from an attacker, another can try by approaching from a different angle. This redundancy of abilities located in the team players allows for a more dynamic gaming strategy.

RAID arrays are becoming the standard storage architecture for servers and other computers that require high reliability. With hardware prices decreasing, RAID systems can store mass data redundantly on multiple disks and exchange data with each other to back each other up. They can be repaired or replaced without halting a system and without loss of information.

In a similar vein, IBM is investing US $1 billion to develop *autonomic computing*: "a systemic view of computing modeled after a self-regulating biological system." 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 will become avail-

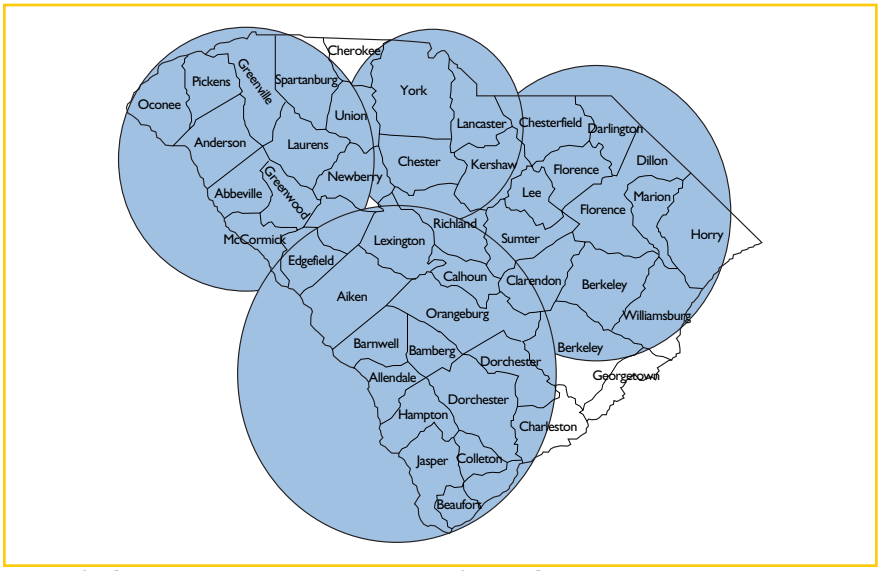

Figure 2. Cell phone tower redundancy in South Carolina.

able for customers in business transactions (see Autonomic Computing, www.research.ibm.com/autonomic/). Signal towers (such as for cell phones) have long used redundancy in providing uninterrupted service. A tower's range often overlaps another tower. Without such overlap, there would be dead spots in signal coverage. Figure 2 demonstrates the redundant nature of this overlap.

Testing Robustness

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.[3] As long as there is sufficient communication bandwidth to find and use healthy resources, it can tolerate the defects. Allowing so many defects lets designers build the computer cheaply.

A research team at the University of South Carolina is investigating the scalability of a system of medium-complexity, heterogeneous agents. The agents form geometric shapes on a 2D grid by communicating with nearby agents. Although only 60 agents are involved, different people built the individual agents.[4] The specifications for the agents were loosely articulated. Working individually, the agents lacked any cohesiveness in forming geometrical shapes; however, when redundant decision-making was introduced, the agents began arranging themselves into the appropriate shapes.

Implications for Developers

Producing robust software has never been easy, and the approach recommended here would dramatically affect 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.
- We need to develop agent organizational specifications 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.

Conclusion

Beyond robustness, we need to make sure not only that our software systems don't crash, but also that they can be trusted. In the next issue, we will describe how agents are the right building blocks for constructing trustworthy systems. These two thrusts—robust software and trusted autonomy—represent the future for agent technology and for software engineering.

Acknowledgement

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

References

1. F.G. Stremler, *Introduction to Communication Systems*, Addison-Wesley, Reading, Mass., 1977
2. D.P. Tollefsen, *Collective Epistemic Agency*, doctoral dissertation, Dept. of Philosophy, Ohio State Univ., Columbus, Ohio, 2002.
3. P. Coffee, "Perfect Computers Cost Too Much," *PC Week*, 6 July 1998, p. 54.
4. V.T. Holderfield, "*A Foundational Analysis of Software Robustness Using Redundant Agent Decision-Making*", tech report, Center for Information Technology, Univ. of South Carolina, Columbia, 2001.

Michael N. Huhns is a professor of computer science and engineering at the University of South Carolina, where he also directs the Center for Information Technology.

Vance T. Holderfield is a PhD candidate in computer science and engineering at the University of South Carolina, where he also is learning to scuba dive.