Stressed Out Systems

Do you manage software? Perhaps you develop software? Then you already know what stress is. As a former developer, program manager, and software engineer, I understand the stress of writing, developing, and managing software systems. There is stress in gathering requirements, identifying users, designing the system, writing the code, and managing changing requirements. There is stress in testing the code, and—once delivered—there is stress in managing the inevitable change upon change upon change.

Stress refers to the pressure, pull, or force exerted upon an object. The resilience of an object, therefore, refers to its ability to recover from stress. Other sources refer to the resilience of an object as its ability to adjust to stress. There are interesting parallels in comparing software to the brain. In fact, the well-studied field of Psychological Resilience is a good starting point.

Paraphrased from Wikipedia:

Psychological Resilience refers to, "The idea of an individual's tendency to cope with stress and adversity. This coping may result in the individual 'bouncing back' to a previous state of normal functioning, or simply not showing negative effects. Another more controversial form of resilience is sometimes referred to as 'post-traumatic growth' or 'steeling effects' wherein the experience of adversity leads to better functioning (much like an inoculation gives one the capacity to cope well with future exposure to disease). Resilience is most commonly understood as a process, and not an individual trait."

Question 1: How do you make resilient software? Answer: you write resilient code by starting out writing non-resilient software, and learning how to keep it running. As part of my software engineering class, my students have to write a bulletproof program (typically, a simple one that prompts for names, hours worked, and hourly rate, and then prints out a simple payroll). I warn them that I will actively try and crash it. Even knowing that I plan on being malicious—I usually manage to crash about 50% of the programs. I run them in front of the class, and ask the class to join in and help me find and exploit flaws. Students initially are somewhat proud of their code, then watch in dismay as I find inputs that will crash their code: invalid inputs, extremely large numbers, zeroes for all inputs, strings for numbers, or very large strings. It is usually their first experience with actively evil input. They learn. They learn to bulletproof their code, to check all inputs, and to test for valid inputs all the time. They learn to trap and handle exceptions. And the viewpoint of writing really resilient good code is learned. You learn to write good resilient code by writing bad resilient code—and improving it over and over (…and over). And then you learn to write code that, when presented with inconsistent or invalid conditions, gracefully recovers, and returns to a consistent and usable state, without destroying data and without invalidating previous work.

Question 2: How can you maintain "normal functionality" in software? Answer: by taking economically reasonable steps to ensure that the user can perform normal operations under almost any type of system stress. In Question No. 1, it was the code that needed to be good. However, in this question, you see that your control over the environment needs to be good, too. Network down? You better have some local cached data to permit emergency functionality. Is the network really slow? Maybe have a good pre-fetch to reduce network latency. Worried about Denial of Service because of overloading or attacks? Use firewalls, redundancy, multiple servers, honeypots, etc. Do you have a single point of failure when contacting remote devices? Maybe you need to have multiple redundant routes to reach them. Mind you; you just can't throw hardware at the problems—you have to analyze the needs of the user, evaluate how the environment will be compromised, and take economically feasible preventative actions to minimize or prevent compromise. Assume your system is constantly under attack—and write not just good but defensive code. In my classes on Enterprise Security, students learn that paranoia is a good trait for network administrators. They are out to get you.

Question 3: How do you get a system to bounce back from failure? Answer: you need to have a process in proactively updating your system in response to constantly changing environments and conditions. Every day there is a new onslaught of viruses, hacks, threats, system vulnerabilities, etc. You cannot just write a program and expect it to be resilient for very long. It takes proactive planning and constant work. It is a continual process, not a single effort. One of the traits of a cyber system is a high degree of interaction between your computer hardware and other physical elements. These physical elements can be networks, remote hardware, and a large collection of physical devices. Cyber systems try to control all of this, and at the same time possibly interact with many other systems. Cyber systems sometimes need extremely high levels of reliability, precision, and coordination among the components—think air traffic control, unmanned vehicle operation, robotic surgery, and healthcare monitoring. Every piece you add gives yet another opportunity for the overall system to exhibit negative behavior (a nice euphemism for fail). There is no sane way to approach this as a single software-writing exercise performed as a solo exercise. You need a high-integrity process to create and update the software. Complex systems require complex processes—processes that are comprehensive, tested, and updated frequently. They need processes that are continually updated as new weaknesses or deficiencies are found.

I never said it was easy. In fact, developers agree—this is hard work. Creating reliable, resilient, robust, high-integrity cyber systems is probably one of the hardest development efforts in the field of software engineering. It is hard to do.

On the other hand, it is a lot easier than living with the potential consequences of not doing it.

David A. Cook, Ph.D.
Stephen F. Austin State University
cookda@sfasu.edu