Software Fault Tolerance at the Operating System Level

Luke Young
Tandem Computers

Providing software fault tolerance at the O.S. level presents some interesting challenges. In designing a fault-tolerant commercial computer system, there can be software-implemented and hardware-implemented tolerance of both hardware faults and software faults. To implement fault tolerance, the kernel should have fault detection, fault diagnosis, fault containment, and fault recovery mechanisms built into it. Another challenge is answering design questions like, "If a system crashes (and reboots) in the field and nobody notices, has it really failed?" (That depends on the application. If the only requirement of the application in question is conserving data integrity, then the answer is no, thanks to wetware fault tolerance.) The issue of system availability (i.e. the portion of time spent in a "good" state as opposed to an "error" or "recovery" state) must also be addressed in designing for Software Fault Tolerance. Reason: For every software application, there exists some non-zero maximum time interval such that all experienced "outage" times less than this maximum are completely acceptable (i.e. "tolerated"), no matter how frequently these outages occur. For some applications, however, the frequency of outages will make a big difference in just how acceptably short these outages must be.

An approach being taken by Tandem to maximize system availability is to classify all possible system outages into one of five types: Physical, Software Design, Operator, Environmental, and Reconfiguration outages. By studying where a customer might experience the most system outage minutes, we can work toward solutions that reduce the durations of these outages. Evidence so far indicates that planned outages make up the bulk of outage minutes, especially in the Reconfiguration and Operator outage classes. Of the unplanned outages, Software Design outages and Operator outages contribute the most outage minutes, so design improvements should focus on these areas.

By considering a kernel as a single component and applying availability modeling and analysis, we have been able to mathematically show that the availability of the kernel can be increased by using several key strategies. One strategy is to increase the fault detection rate by decreasing the mean time to error detection. This is being implemented in NonStop Unix through kernel features such as assert checking, data structure audits, stack overflow detection, and a "deadman timer." Another strategy is to increase the recovery rate by reducing the duration of recovery and increasing the probability of making a true recovery. This is being implemented through features such as assert failure recovery, subscription services, and a fast reboot / fast crash mechanism. Yet another strategy is to decrease the "false recovery" rate by reducing both the duration of recovery and the probability of false recovery by providing recovery in stages.

These explored paths are far from being fully traveled. In practice, for example, it may be limiting to consider a running kernel as a single component. Fault tolerance for a kernel may not mean fault tolerance for a particular process or application. As software fault tolerance matures, it would be nice to see high levels of availability attained - not just at the system level but at more and more subsystem levels as well.