



HA designers think outside the box

By *Paul N. Leroux*

Studies indicate that software-related issues, including software errors and software upgrades, account for most service outages. Paul makes the case in this article that achieving HA must, as a result, start with the software foundation on which all HA applications are built: the OS.

A systems problem

There's no shortage of High Availability (HA) products on the market, many of them promising 5-nines or greater reliability. Question is, can HA really come out of a box?

The numbers tell the story. To achieve 99.999% reliability, an application can experience no more than 5.25 minutes of downtime a year – it doesn't matter how many software faults, power outages, hardware failures, operator errors, or other gremlins pop up. In fact, if an application were removed from service for just one hour a year for software upgrades, it would achieve a miserable 4-nines availability rating: 99.99 percent.

With numbers like these, it soon becomes obvious that HA is, in fact, a systems problem. The entire solution, including the Operating System (OS), hardware platform, and application code, must be designed with HA in mind. For instance, how does the OS handle driver faults, a common cause of system crashes? Can the OS immediately restart the rogue driver without a system reset? Or must the entire system be rebooted?

Even physical security comes into play. If a system is located where unauthorized personnel can physically tamper with it, maliciously or otherwise, it can't realistically qualify as an HA system. HA can also extend to operator training and human factors engineering. If a system failure requires manual intervention (for instance, replacing a faulty CPU card), operators must learn how to solve the problem promptly, and the system itself must be designed to make the repair job easy.

Even if an out-of-the-box solution could provide HA, it might not be the kind of availability the user requires. To understand, consider that an HA design does more than simply try to prevent failures. It must also recover from any error or system upgrade within a very brief (and hopefully predictable) time frame. This can be expressed in a simple formula (see Formula 1) where *A* is availability, *MTTF* is Mean Time to Failure (average length of time the system will operate before failing), and *MTTR* is Mean Time to Repair (average time it takes for the system to return to service after any component fails or is upgraded):

$$A = \text{MTTF} \div (\text{MTTR} + \text{MTTF})$$

Formula 1

From this, we can deduce that system A could fail more often than system B, yet both systems could have the same availability rating, provided that A had a shorter MTTR. But would system A be more of a nuisance or a danger to users? It all depends on the application, of course. But, clearly, to achieve a meaningful degree of HA, the systems designer must first understand what kind of service (and service outages) users prefer, then design the system accordingly.

Nor is an HA rating meaningful until you look at how partial service outages are handled. For instance, say the performance of a network degrades to the point where transmission quality becomes unacceptable. The system may be operational, but is it truly available to the customer? Not really. Actually, this brings up another point: network management – along with technologies such as redundant or load balancing network links – can form yet another part of the overall HA equation.

Software: the biggest challenge

If hardware were the only consideration, most systems designers could achieve HA by adhering to a few time-tested principles:

- Prefer circuitry to mechanics.
- Avoid fans by selecting cool-running devices.
- Choose components designed for easy repair or replacement.
- Look for rugged construction.

In fact, a rich variety of "HA" hardware platforms that embody these principles can be purchased off the shelf. Many of them are based on CompactPCI or AdvancedTCA, both of which support a number of HA-related features, such as hardware hot swap for dynamically replacing obsolete or faulty components.

The biggest problem, however, is software. Studies indicate that software-related issues, including software errors and software upgrades, account for most service outages. Achieving HA must, as a result, start with the software foundation on which all HA applications are built: the OS.

Driver, heal thyself

To appreciate the role of the OS, consider the principle of fault isolation and recovery – a key concept in HA. According to this principle, a fault in any software or hardware component must not throw other components into error states; otherwise, system-wide failure may occur. Moreover, the faulty component must be quickly replaced or restarted, while the rest of the system continues to run.

This requirement poses an inherent problem for conventional RTOSs, since they typically run most or all software components in the same memory address space as the OS kernel. In this architecture, a corrupt pointer in even a trivial component can overwrite kernel memory and thereby crash the entire system. So much for fault isolation.

Some RTOSs attempt to address the problem by running applications in separate, memory-protected address spaces. If an application attempts a memory violation, the MMU will catch the fault and alert

the OS, which can then take appropriate action (for example, slay the offending process, then restart it). Unfortunately, these OSs still bind drivers, protocol stacks, file systems, and other system services to the kernel, which means that any of these components can induce a kernel fault. In other words, each driver or protocol stack remains a *Single Point of Failure* (SPOF) – a component whose failure can render the entire system unusable.

Other OS architectures, such as the microkernel architecture used in the QNX Neutrino RTOS, go one step further and allow every system service to run in its own MMU-protected address space. Faulty drivers and protocol stacks no longer act as SPOFs, but instead can be stopped (and automatically restarted) before they cause other services to fail. See Figure 1.

Fixing it before it breaks

This last type of OS architecture enables another key requirement of HA: the ability to repair software errors before they have a chance to recur. For instance, if a driver attempts to write to memory allocated to another process, the MMU will notify the OS, which can then pass control to a program called a software watchdog (a.k.a. process monitor). The watchdog can then do two things:

1. Restart the driver and, if required, any related processes.
2. Generate a process dump file that can be analyzed with source-level debugging tools.

Using this dump file, the software developer can immediately identify the source line that caused the fault and inspect diagnostic information such as the contents of data items and a history of function calls. With this information in hand, the developer can quickly engineer a fix that can be uploaded to other units in the field, often before they experience similar failures. See Figure 2.

Better still, a software watchdog can be used to monitor system events that are invisible to a conventional hardware watchdog. For example, a hardware watchdog can ensure that a driver is servicing the hardware, but may have a hard time detecting whether other programs are talking to that driver correctly. A software watchdog can cover this hole and

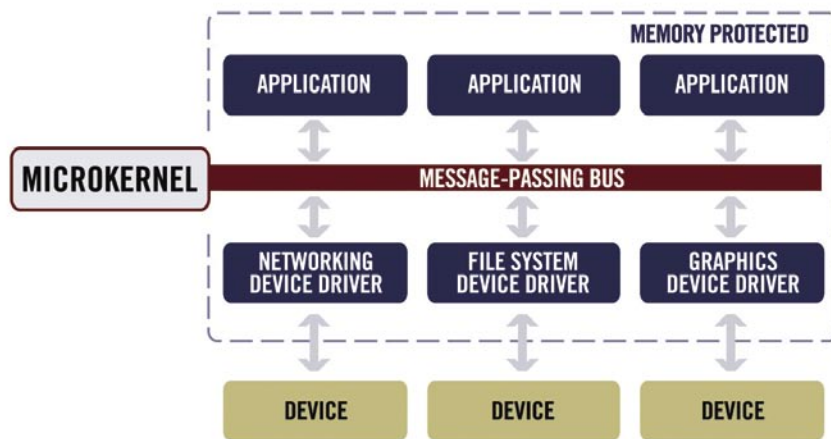


Figure 1

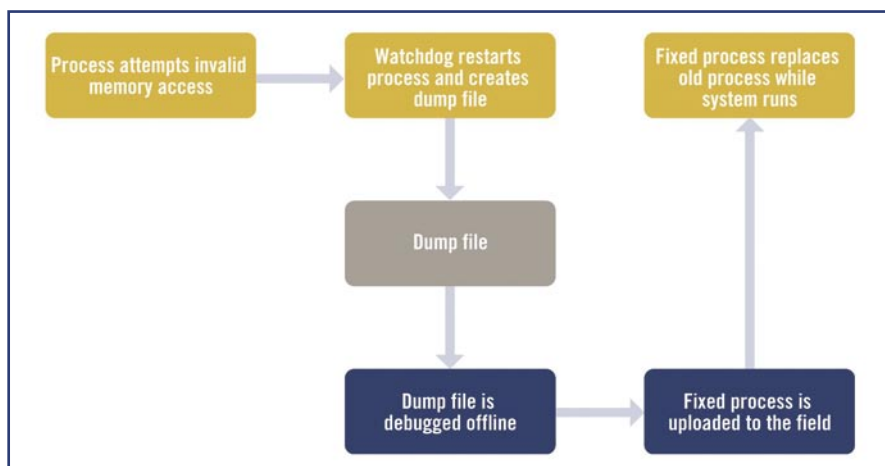


Figure 2

take action before the driver itself shows any problems.

A software watchdog may offer a number of other features, such as:

- Heartbeating – Allows the watchdog to monitor the progress of any software component and to detect problems before they escalate to an unrecoverable state.
- Resilience to internal failures – If the watchdog stops abnormally for any reason, it should be able to immediately reconstruct its own state.
- Watchdog API – For telling the watchdog what actions it should take should an error condition occur.

Making the (up)grade

A higher-end networking element or industrial control system may have to be upgraded with new software several times a year. In most such systems, downtime isn't an option: the system must continue

to provide service while the upgrades take place. As with other HA software issues, the responsibility for enabling such upgrades falls squarely on the OS.

When it comes to upgrading application-level code, most modern OSs have little problem. In fact, some OSs even allow new system services, such as drivers and protocol stacks, to be dynamically attached to the kernel. However, because these services then run in kernel space, they can't easily be stopped, removed, and replaced with new versions. Upgrading them becomes difficult, if not impossible, unless the system is removed from service and rebooted.

To address these problems, an OS should, at a minimum, allow drivers and other services to be dynamically unloaded. However in many cases a driver may have to be upgraded without interrupting the service that the driver itself provides. As a result, the OS should allow a new version of the

driver to start while the old version is still running, and then allow the new version to gracefully take over the existing driver's duties. Once the transition is complete, the OS could terminate the old driver and recover whatever resources it was using.

A matter of design

Designing for HA requires a split personality. The systems designer must first ensure that faults – in both hardware and software – happen as rarely as possible. But the designer must also assume that faults will occur and take precautions to ensure that the system recovers quickly.

The question is: which precautions are necessary? To answer that, the designer must first determine which services in a system actually require HA and what degree of HA each of those services needs. Once that has been decided, the designer can begin to identify any potential SPOF, that is, any component – be it a CPU, networking card, power supply, or software module – whose failure can impede or stop delivery of the service.

Redundant choices

Having identified potential SPOFs, the designer can decide on what kind of redundancy, if any, is required. There are two basic choices: N redundancy ($2N$, $3N$,...) where every component is duplicated, or $N+1$ redundancy where only selected components have a backup. A $2N$ design can often provide faster switchover times than an $N+1$ design but can also be much more expensive if, for instance, the system has a large number of I/O connections (as in a switch). In that case, a single spare connection could serve as the standby for several active connections.

Redundancy may be extended to any resource, including, for example, network

links. The redundant links could act as standbys or provide load balancing for greater throughput. See Figure 3.

Besides deciding on the form of redundancy, the systems designer must establish what restart model the redundant components will use. For instance, in a $2N$ configuration, the redundant system can act as a hot standby (fully initialized backup), warm standby (partially initialized backup), or cold standby (uninitialized backup). A hot standby design can provide the fastest recovery but can also be more complex to design, since the standby node must maintain complete, up-to-the-moment state information about system activity.

In fact, design problems go beyond the redundant components themselves and extend to any applications communicating with those components. For instance, let's say a node provides compute services for client applications throughout the network. What happens if that compute node fails and another node, acting as a hot standby, takes over? Suddenly, those client applications must now all talk to a different node (the standby system).

With most OSs, achieving this *failover* can require careful design and network-specific code. But if the OS provides network-transparent Interprocess Communication (IPC), where applications don't effectively have to know which node they are talking to, then much of the effort and complexity of switching over to the redundant node disappears. In a distributed operating system like QNX Neutrino, for instance, accessing resources such as disks, file systems, and I/O ports on other nodes is so transparent that an application would need special code to know which node a resource resides on. From the application's perspective, then, there would be no real difference between talking to a primary or standby computer. See Figure 4.

Using more than one basket

The systems designer can also eliminate SPOFs by distributing applications and services across a cluster of loosely coupled CPUs, whether those CPUs reside in the same chassis or in physically separate machines. With a cluster, most services can continue running even if one node experiences catastrophic failure – it's a lot like putting your eggs in more than one basket. Clustering can also yield greater through-

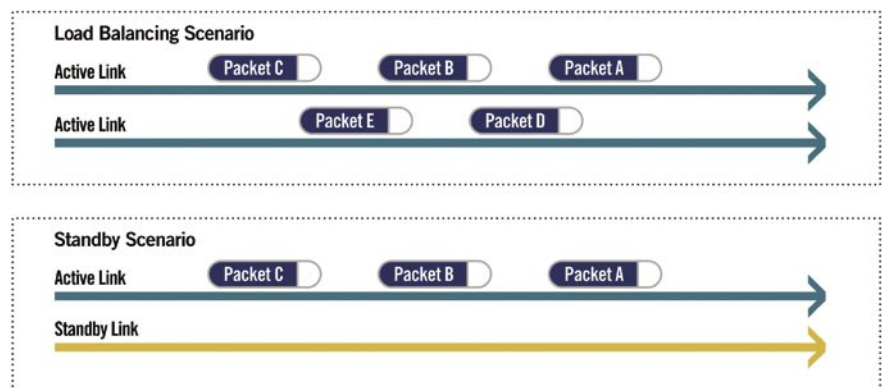


Figure 3

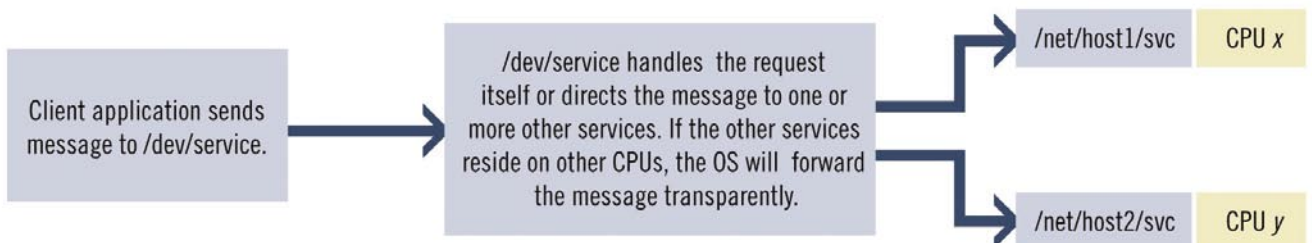


Figure 4

put, since each node adds its own CPU power, memory, I/O bandwidth, and so on.

Still, deciding which applications and hardware resources should be allocated to which node is a challenge all its own. In fact, it may be difficult to know until the integration phase whether processes and peripherals have been distributed in a way that provides optimal performance. The job is simplified, however, if the OS provides network-transparent IPC. For instance, applications could continue to access a disk drive even if that drive were moved to another node; no recoding would be required.

Out of the box?

HA demands a systems approach that encompasses everything from properly inspected solder joints to well-trained system administrators. Nonetheless, for most mission-critical applications, the cost of implementing HA is typically far less than the cost of the alternative: downtime. That's especially true today as more OS, board, and system vendors deliver HA features, thereby saving the systems designer from doing everything from scratch. HA itself may not come out of the box, but many of the tools to achieve it can.

Paul Leroux is a technology analyst at QNX Software Systems, where he has served in various roles since 1990. His areas of focus include OS architecture, high availability systems, and integrated development environments.

For further information, contact Paul at:

QNX Software Systems

175 Terence Matthews Crescent
Ottawa, Ontario • Canada, K2M 1W8
Tel: 613-591-0931
E-mail: paul@qnx.com
Website: www.qnx.com