12367.65234.0240
32163912.921.0321389
42738823.921.0321389

239923.09348723.942
23894239.3928492

329849423.29300
3427284.3204900

234989234.0240

12367.65234.0240
32163912.921.0321389
42738823.921.0321389

239923.09348723.942
23894239.3928492

329849423.29300
3427284.3204900

234989234.0240

287283782.0302.3290

2938498243.0390.9384
438043209.03219034

84923.938249567
209765.759656

699489.6989

565.59689567

5690795

5609

4

A few early steps show there's a long (and bumpy) road ahead.

12367.65234.0240
32163912.921.0321389
42738823.921.0321389
239923.09348723.942
23894239.3928492
329849423.29300
3427284.3204900
234989234.0240
287283782.0302.3290
2938498243.0390.9384
438043209.03219034
84923.938249567
209765.759656
699489.6989
565.59689567
5690795
5609
4
234989234.0240
287283782.0302.3290
2938498243.0390.9384
438043209.03219034
84923.938249567
209765.759656
699489.6989
565.59689567
5690795
5609
4

MICHAEL W. SHAPIRO, SUN MICROSYSTEMS

Self-Healing

in Modern Operating Systems

Driving the stretch of Route 101 that connects San Francisco to Menlo Park each day, billboard faces smilingly reassure me that all is well in computerdom in 2004. Networks and servers, they tell me, can self-defend, self-diagnose, self-heal, and even have enough computing power left over from all this introspection to perform their owner-assigned tasks.

Then, after arriving at my office, I reacquaint myself with reality: that every IT manager, system administrator, and developer is fighting against the monster of computing complexity. The worst possible situation to be in is trying to identify, root-cause, and resolve a problem in today's complex stack. Regardless of who their vendor is, the administrators I talk to around the world don't look much like the ones on the billboard, who seem like the only other thing they need from their server is perhaps a martini dispenser.

While we need no reminder of the cost of complexity to the industry, it is worth wondering: Where are we *really* on the road to self-healing systems? How much of the problem is still open research versus lack of execution or priority on the part of vendors? Are we making more progress in hardware or in software? And how useful a solution can we expect, given the software we have now

Self-Healing

in Modern Operating Systems

versus needing it to be modified or rewritten?

In tackling these issues head on for Sun's upcoming Solaris 10 release, I've come to see that while the industry is gaining focus on the right set of problems, the reality is far behind the hype and we have a long road ahead.¹ Increasingly, there is a need for designers of operating systems and other system-level components that sit between hardware, applications, and administrators to play a vital role in facilitating a real leap forward in self-healing computer systems.

THE ROLE OF THE OPERATING SYSTEM

There are three basic forces we can bring to bear on improving the availability of computing services: we can improve the reliability and resiliency of the individual components (hardware or software); we can introduce redundancies to cope with component failures; and we can predictively avoid failure or reduce the time required to recover. Yet some important trends in the industry are largely at odds with the desire to increase component reliability and redundancy. First, there is the growing use of commodity hardware components from disparate sources to build cheaper systems. Similarly, modern software stacks are being constructed from commodity components as well—in many cases, from open source or off-the-shelf components of widely varying quality. Second, the desire to increase redundancy is often at odds with the need to reduce the cost, management difficulty, and complexity of the solution while maximizing its overall performance.

So while improvements in these first two areas are important to any overall solution, the usefulness of self-healing systems is fundamentally about making significant progress in the third area: reducing recovery time and implementing systems that can diagnose, react to, and even predict failures. Aaron Brown, David Patterson and others have similarly suggested focus in this area as a means of making more significant progress in overall availability.² Yet the complexity of the universe that a self-healing system must understand to make effective decisions is rapidly increasing. Even a basic system or blade will soon have multiple processor cores per die and

multiple hardware threads per core: Sun, AMD, IBM, and Intel are all hard at work here. Even a service deployed on a single system is of increasing depth and complexity: multiple threads per process, multiple processes per component, and a variety of components from different authors stacked on top of each other.

In this emerging world, each system will be able to do more useful work by supporting more application services with more memory, compute power, and I/O. One approach is simply to make an individual system the unit of recovery; if anything fails, either restart the whole thing or fail-over to another system providing redundancy. Unfortunately, with the increasing physical resources available to each system, this approach is inherently wasteful: Why restart a whole system if you can disable a particular processor core, restart an individual application, or refrain from using a bit of your spacious memory or a particular I/O path until a repair is truly needed? Fine-grained recovery provides more effective use of computing resources and dollars for system administrators and reduces downtime for end users.

Self-healing functionality for users and administrators of a modern operating system will provide fine-grained fault isolation and restart where possible of any component—hardware or software—that experiences a problem. To do so, the system must include intelligent, automated, proactive diagnoses of errors that are observed on the system. The diagnosis system must be used to trigger targeted automated responses or guided human intervention that mitigate a specific problem or at least prevent it from getting worse. Finally, these new system capabilities must be connected to a new model for system administrators oriented around simpler, higher-level abstractions.

Your operating system provides threads as a programming primitive that permits applications to scale transparently and perform better as multiple processors, multiple cores per die, or more hardware threads per core are added. Your operating system also provides virtual memory as a programming abstraction that allows applications to scale transparently with available physical memory resources. Now we need our operating systems to provide the new abstractions that will enable self-heal-

ing activities or graceful degradation in service without requiring developers to rewrite applications or administrators to purchase expensive hardware that tries to work around the operating system instead of with it.

A key requirement of these abstractions, however, is that they enable self-healing systems to make diagnoses and take actions that, like those of your human doctor, should begin by doing no harm. The operating system and self-healing software can implement intelligent self-diagnosis and self-healing only if they understand significantly more about hardware/software dependencies than they have historically, and more about the relationships and dependencies in the software stack deployed above. To see why, let's consider a simple example.

The kernel can detect the failure of any running process by handling various types of exceptions and deciding to terminate the process or pass an exception along to it (in Unix terms, it can send the process a signal such as SIGSEGV or SIGBUS). These exceptions usually cause the process to terminate by default, but can also be intercepted by more intelligent applications to try to clean up or save data before dying. Historically, such signals indicated a programming error; the errant process attempted to read from an unmapped or misaligned address (for example, dereferencing a null or bogus pointer). On a modern system such as Solaris, however, you can also detect and attempt to recover from an exception where a process accessed memory with an underlying double-bit ECC (error correcting code) error, which can be detected but not corrected by the typical ECC code protecting your DRAM. Similar scenarios can occur with errors in the processor core itself and its L1 and L2 caches, with varying degrees of recovery possible depending on the capabilities of the underlying hardware.

Now the big question: Is it still safe to signal or termi-

nate the process? And if you terminate the process, can you restart it? How? What else might be affected? What of this do we need to explain to the administrator? In other words, what's a self-healing system supposed to do?

To illustrate the complexity of the problem, let's consider a few quick scenarios related to only the first question. If the affected memory region is not shared and the process has no relationship to other processes on the system, then we can terminate the process and simply stop using the affected physical page of memory rather than returning it to use in the kernel's free page list. As we observed earlier, however, most processes aren't like that anymore. If you simply terminate the process, then its sudden death may cause a portion of a multiprocess application suddenly to go missing, causing the application to deadlock or misbehave. Or the process may be providing some type of service to other processes (e.g., a name service, a database back end), which would cause a cascading failure in other applications.

If you send the process a signal, you may also be in trouble; if the signal handler contains code that accesses the same bad piece of memory while trying to print a message or save data while cleaning up, the same error can recur. Any signal that results in a core dump could prove confusing to administrators—the software in this example is an innocent victim of a hardware problem, and no one should waste time attempting to debug this particular core file or application software code.

Finally, if the error is in a shared memory region, things get even worse. Many modern multiprocess applications contain their own restart capability, wherein a parent process monitors and restarts its children. If a child process died from touching shared memory and was restarted, the new child might well immediately touch the same location again and repeatedly die, again causing

application failure or serious degradation of service. From these few scenarios, you might be tempted to say, "Pass enough information to the application to let it decide what to do," and punt the problem to application developers or administrators. This leads us back to one of the original questions: Is the next generation of self-healing technology going to require everyone to rewrite

Self-Healing

in Modern Operating Systems

their applications? Handling that kind of signal sounds complicated, prone to bugs, and not particularly portable.

Meanwhile, what about the underlying failure in our processor or memory? How are we resolving that? And what is the administrator supposed to see? Our self-healing system should not be saying, "I killed process-ID 123 because of a fatal ECC error at physical address 0x12345678." Administrators need to know what hap-

pened in terms they can understand: what *service* was affected and how does that affect other services; what was the impact (performance, functionality); what did the system do for us (restarting something, failing over, using a redundancy); and what, if any, human intervention is needed (Do we need to replace the DIMM yet? How soon? Which one?).

These examples emphasize the need for new abstrac-

tions in the system that can solve these problems without continuing to overburden developers and administrators. They also demonstrate that we need to make progress in two distinct areas: the ability to implement rapid, intelligent responses to errors that are detected in the system as they happen, and the ability asynchronously to diagnose observed errors to underlying problems. Once a self-healing system has diagnosed a faulty hardware component or broken application, it can use this knowledge to trigger actions, such as disabling a component, failing over to a redundancy, or notifying a human administrator that a repair or patch is needed.

SELF-HEALING SYSTEM MODEL

A self-healing system is one that:

- Replaces traditional error messages with robust error detection, handling, and correction that produces

telemetry for automated diagnosis

- Provides automated diagnosis and response from the error telemetry for hardware and software entities
- Provides recursive fine-grained restart of services based upon knowledge of their dependencies
- Presents simplified administrative interactions for diagnosed problems and their effects on services and resources

To make tangible progress on these problems, at Sun we adopted a two-pronged approach to improving availability in Solaris 10: we implemented an extensible *fault manager* that can receive telemetry from system components, including the kernel, and pass them to self-healing diagnosis and response software; and we implemented a *service manager* that manages descriptions of the services on the Solaris system and their interdependencies, and can implement intelligent automated restart. Finally,

Self-Healing

in Modern Operating Systems

we introduced the notion of a new operating system primitive called a *contract* that allows a service manager to describe expectations for recovery of a set of managed processes to the kernel. I'll discuss them in turn to illustrate the benefits and challenges of each, and show how they work together to make a useful first step toward self-healing.

We believe that our model extends to any system or to hierarchical networked compositions of systems. A fault manager is itself a service that receives incoming error telemetry observed by the system and uses appropriate algorithms or expert-system rules to attempt to diagnose these errors automatically to an underlying problem, such as a hardware fault or likely defect in an application. A service manager manages the various application services running on the system and uses their dependencies to implement orderly startup, shutdown, and restart. So while the Solaris fault manager and service manager deal with local resources such as CPUs, memory, I/O devices, and single-system services, we believe the same concepts would apply when designing self-healing features for a rack of blades or a networked data center, where a fault manager would track the list of known service outages and a service manager would observe and manage the highest-level set of services offered to the network or data center as a whole.

FAULT MANAGER AND DIAGNOSIS

In our self-healing design, the fault manager is responsible for implementing asynchronous automated diagnoses of problems from error symptoms. It then uses the results

of each diagnosis to trigger an automated response, such as offlining a CPU, device, region of memory, or service, or communicating to a human administrator or higher-level management software. The fault manager therefore manages the list of *problems* on the system and exports this as its abstraction to human administrators and higher-level management applications, rather than the individual underlying error messages it has received. While we believe this new abstraction layer will significantly reduce complexity, it also forms the basis of a major change from the traditional error model.

The original Unix design was not concerned with mechanisms for hardware or software fault-handling: complex error-handling routines and fault-diagnosis capabilities would have compromised its simple elegance and portability. So even a basic concept like error logging, provided in Unix by the syslog service developed in the 1980s, is little more than interprocess or networked

Hierarchy of Processes and Restarters

FIG 3

printf() and has barely evolved since then. Yet this model has proved to be an enduring and tantalizing trap of sorts for developers: in some sense, it will always be easier to write an application that simply emits random printf() strings for humans than to think deeply about failure semantics and data for automated diagnosis. Most often, administrators are left trying to awk, grep, and Perl their way through error logs to understand the problem.

Unfortunately, building self-diagnosing systems on top of undecipherable and bizarre error messages from off-the-shelf components is likely to produce something ineffective in diagnosing real problems, brittle in the face of changes to the software, or both. Often, even if a message seems friendly to humans, the raw information that would be needed to drive an intelligent automated diagnosis system is simply absent. In Solaris 10, we preserved the traditional syslog facility, but created a new information channel for structured telemetric events, described by a simple and extensible protocol, to be passed from the kernel to the Solaris fault manager. In other systems where a structured error event mechanism is already present, developing fault managers as clients of the existing mechanism may make more sense. System designers must also decide whether error telemetry is actually removed from the administrative interface or not when automated diagnosis is provided. Removing human-readable error messages as part of the conversion to automated diagnosis simplifies user interactions, but may also break legacy scripts and expectations for users who have extensive experience with old behaviors.

The fault manager, shown in figure 1, logs each event and passes the telemetry on to an appropriate *diagnosis engine*, a small software module that attempts to diagnose the problem and then trigger an automated response. The final response may in turn include messages back to syslog or to the system console intended for a human, but the underlying errors converted to telemetry are no longer seen as messages (they can be retrieved from the log if necessary). We implemented diagnosis engines for CPU, memory, and I/O components; for software applications we focused on restart, described next.

One example diagnosis engine examines error telemetry sent by the kernel for correctable and uncorrectable memory errors. While correctable memory errors are transparent to applications (typical DRAM ECC codes permit single-bit errors to be corrected and good data returned to the processor), a sequence of correctable errors with particular properties may indicate the imminent or present failure of a DRAM cell, row, column, or connector. Depending on what events are observed, our

diagnosis engine may ask the kernel to retire a particular physical page of memory, remapping it out from under running processes by reassigning a new physical page. This change is transparent to applications; the old data can be copied or reread from the page's backing object if the page is clean. The diagnosis engine may also emit a message for a human, requesting replacement of a DIMM.

The fault manager's design therefore provides a jumping-off point for research in improving the quality and effectiveness of automated diagnosis. Self-healing systems will require significant basic research into methods for effectively diagnosing problems in both hardware and software. We tried to facilitate such work by providing a simple programming model for pluggable diagnosis engines in our system and tools to aid in their design and testing. Experience in the industry suggests a wide variety of approaches can be brought to bear here, including application of statistical methods, expert systems, control theory, biometric computing, and others.

SERVICE MANAGER AND DEPENDENCIES

The service manager is used to enable intelligent, fast application restart following a software error that results in a process core dump or when a process falls victim to an uncorrectable hardware error and must be terminated. Two new features had to be added to Solaris to implement this functionality: first, a way for services to describe themselves and their dependencies to the service manager; second, a mechanism that relates each process to another process that is functioning as its *restarter* and indicates that a process is restartable in the event of an unrecoverable hardware or software error.

Traditional Unix systems use a collection of shell scripts to start services when the system boots; typically these are located in the /etc/init.d and /etc/rc*.d directories. Unfortunately, while each script typically implements a simple start and stop mechanism, they contain no explicit expression of dependencies. The alphabetic ordering of the scripts is used to determine the order in which services are started. For example, /etc/rc2.d/S10foo starts before /etc/rc2.d/S20bar, but there is no way to know whether bar actually depends on foo, or whether the developer simply picked a random name. Even without the need for advanced restart, this mechanism has caused dozens of bugs over the years where incorrect ordering and the introduction of new services caused race conditions and failures in starting up system services. With restart, the situation is hopeless: without dependencies, there is no way to know what scripts to run in what order to do no harm and effect a restart.

Self-Healing

in Modern Operating Systems

The concept of fine-grained restart is the subject of existing research, but it is very interesting to see it realized in a production Unix system.³ We converted all of the services in base Solaris to use our service manager, and a portion of the resulting service and dependency graph is shown in figure 2. Each rectangle indicates a service, and each edge a dependency. Dependencies of like attributes are grouped together and assigned names; the attributes are not shown. This is a powerful illustration of the complex web of dependencies on a modern system before you even start adding anything to it.

The lack of dependencies in the traditional Unix mechanism has also been addressed elsewhere, most notably in NetBSD, where startup scripts were annotated with dependency statements.⁴ Similarly, Windows provides a Service Control Manager that includes service dependencies and can perform restart of individual processes; and analogous mechanisms have been implemented in Sun's Cluster product and by other vendors. In examining these existing approaches and considering our future needs for Solaris, we identified several important design considerations essential to any service manager:

A service manager must provide a means for a service to describe itself and its dependencies to the system.

This can be done by means of programming APIs (as in Windows), shell scripts or agents that accompany the service (as in NetBSD and Sun's Cluster), or meta-data such as an XML file, perhaps in combination with APIs and/or scripts and agents. In Solaris, the existing shell script mechanism was too unstructured for our needs and also not likely to extend well as we enrich service descriptions to meet future needs. We also wanted to adapt existing applications to our service manager without modifying application source code. For these reasons, we chose to define an XML schema and ask developers to bundle an XML file with an existing application as its service description.

A service manager must provide a means for notification of process failure. If support for single-process services is all that is required, then using existing operating system APIs to wait for a process to exit may be sufficient.

If support for multiprocess services with arbitrary descendants is required, a system primitive that records the relationship between a collection of processes and their restarter is required. The notification mechanism must also include information on the failure sufficient to effect an informed restart. In Solaris, many existing services were already multiprocess, so a new system abstraction called a *contract* was developed for notification.

A service manager may optionally provide a means for delegation of restart responsibility to another service.

Delegation can be used to add specialized restarters with differing policies to the overall service restart facility, but may not be required for systems where a single restarter is sufficient. Our service manager supports delegation but requires participating restarters to use a programming API to receive events. This API is used by the Solaris `inetd(1M)` daemon and can be used for cluster fail-over and other purposes.

Most typical existing Solaris services were converted to our service manager in a few hours by writing an appropriate XML file to accompany the service. The XML file describes what scripts or commands implement the start and stop functionality, and includes a structured list of dependencies on files, conceptual milestones in system booting, and other services. We also provide compatible startup and shutdown for services that deliver only legacy scripts. As with error messages, our initial experience in service conversion suggests that while more thinking is required up front by the developer, the result is a more stable interaction with the system that is less brittle in the face of customization and change, and provides benefit to administrators and end users through fine-grained restart that is not otherwise possible.

One of the major consequences of our choices is that by imposing increased order on the system, we require its citizens to abide by some new rules and regulations. For example, the legacy serialized startup allowed developers unknowingly to release applications with incorrect notions of dependency. We discovered several preexisting bugs and problems with dependencies and robustness of various services as a result of converting such a wide

variety of existing applications and parallelizing startup. Therefore, while our new mechanism allows applications to participate without changing their source code, it is clear that software written for a self-healing environment must be tolerant of restart and have a proper description of dependencies. New research and tools will be needed to help application developers write and test for a self-healing environment that includes fine-grained restart.

CONTRACTS AND RESTARTERS

When the Solaris service manager executes the commands that start each service, a new kernel resource called a *contract* is set up to indicate that all descendant processes of this command are restartable and can be terminated on hardware or software errors as long as the operating system sends a notification event to the restarter named in the contract. The contract therefore acts as the mechanism for receiving restart notification events, embodies the relationship between every process and its restarter, and tells the kernel that a process can be terminated on a fatal hardware error, as in our earliest example.

The contract itself appears as a file in the kernel-managed pseudo-filesystem `/system/contract/` where files represent each contract event endpoint. If a restarter itself dies, it can be restarted and reacquire its contracts. At the root of the entire tree of processes is the Unix init process, which in turn is automatically restarted by the kernel as process-ID 1 if it dies. The contract settings determine the behavior for various types of failures: events can be received for software failures, hardware failures, and normal process exit behaviors. Using these events, the service restarter can determine if a process died as the result of a software or hardware failure and react accordingly. Each contract can be configured to terminate all of the processes in the corresponding process group or in the entire contract if any one fails, thereby avoiding deadlocks and inconsistencies in multiprocess applications and permitting the whole service to restart safely.

As discussed in the previous section, the service manager can also *delegate* responsibility for restart to a descendant. Delegation is implemented by permitting the delegated restarter to manage contracts associated with its own descendants. The delegation APIs also permit sophisticated applications to take responsibility for their failure behavior by reacting to events directly, and they allow other kinds of service managers such as clustering fail-over software, batch queuing systems, and grid computing engines to participate in restart.

We use delegation to provide restart for the traditional

Unix Internet services such as telnet, ftp, and rlogin; the `inetd` daemon acts as a delegated restarter for these services, in addition to listening on various sockets. Figure 3 shows an example hierarchy of restarters and processes, including Unix init, the service manager daemons `svc.startd` and `svc.configd`, and others.

CONCLUSIONS

Sun's experience designing availability features for Solaris 10 shows that significant work is needed in creating new abstractions for advancing the state of the art in self-healing by appropriately leveraging the operating system. Only by delivering these new abstractions can we fully benefit from progress in basic research in improving reliability and diagnosis, deliver higher availability through correct recovery and predictive problem avoidance, and provide self-healing that is simple and real for developers and administrators. Q

REFERENCES

1. See <http://sun.com/software/solaris/10/> and <http://sun.com/msg/>.
2. Brown, A., and D. Patterson. 2001. Embracing failure: A case for recovery-oriented computing (ROC). High Performance Transaction Processing Symposium, Asilomar, CA (October 2001); see <http://roc.cs.berkeley.edu/>.
3. Candea, G., and A. Fox. 2001. Recursive restartability: Turning the reboot hammer into a scalpel. Proceedings of the 8th Workshop on Hot Topics in Operating Systems (May 2001); see <http://i30www.ira.uka.de/conferences/HotOS/>.
4. Mewburn, L. 2001. The design and implementation of the NetBSD rc.d system. Proceedings of the 2001 Unix Annual Technical Conference, Boston, MA (June 2001); see <http://www.mewburn.net/luke/papers/>.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

MIKE SHAPIRO is a senior staff engineer and architect for RAS (reliability, availability, serviceability) features in Solaris Kernel Development. He led the effort to design and build the Sun architecture for Predictive Self-Healing, and is the co-creator of DTrace. His contributions to Solaris include the DTrace D language compiler, kernel panic subsystem, `fmd(1M)`, `mdb(1)`, `dumppadm(1M)`, `pgrep(1)`, `pkill(1)`, and numerous enhancements to the `/proc` filesystem, core files, crash dumps, and hardware error-handling on Solaris. He holds an M.S. in computer science from Brown University.

© 2004 ACM 1542-7730/04/1200 \$5.00