Solaris Service Management Facility: Modern System Startup and Administration

Jonathan Adams, David Bustos, Stephen Hahn, David Powell, and Liane Praza
– Sun Microsystems, Inc.

ABSTRACT

Application uptime is critical to every administrator. The factors which cause system downtime are often handled by the operating system, but causes for application faults (e.g., software bugs, hardware faults, or human errors) are not addressed by standard system software. Recovery is left to humans, who may often compound the problem due to misdiagnosis or simple error. While availability issues have traditionally been addressed by expensive high-availability clustering solutions, the increasing complexity of software stacks requires a solution for all systems.

In addition to the challenges of managing availability of higher level software, the modern operating system itself is composed of many interdependent software entities. A failure in any one of these components often cascades, causing failures in other components. A complex software model with many interdependent elements makes diagnosing failures very challenging for system administrators. The traditional `init.d` script mechanisms for UNIX are only a weak reflection of the intricate dependency relationships which exist on every system.

We introduce the Service Management Facility (SMF) as a comprehensive way to describe, execute, and manage software services. SMF promotes the service to a first-class operating system entity, without requiring modification of application binaries or changes to the UNIX process model. It relieves the administrator from duties of application failure detection and restart, and provides sophisticated diagnosis tools when automatic repair is impossible.

Introduction

As software running on a system becomes more complex, the traditional separation of management of system startup from run-time management becomes untenable. The reliability of modern hardware, coupled with the ever-increasing complexity of modern software means that applications are just as likely, if not more likely, to be the point of failure on a system. Reboots are costly, especially on sophisticated hardware.

A system administrator is expected to ensure that critical applications are always running. This is sometimes done by manual monitoring or an expensive to maintain home-grown tool, or the administrator is expected to turn to software not well integrated with the operating system – either sophisticated monitoring packages, or expensive and often complex high-availability clustering software. The unbundled solutions usually also focus on higher level applications, and can neglect potential failures of core operating system daemons.

But, even higher level applications do not stand on their own. In order to provide business-critical functionality, often two or more pieces of software must be working and cooperating closely together. Failure of even a less critical application can have unforeseen consequences in important software.

In addition to basic availability concerns for software services, the service management model needs to be significantly enhanced. Without a common model and interface, it is very difficult to ask the system even basic questions such as: "what's running?", "what's broken?", and "what applications are available on this system?".

Application management is a fundamental task of all administrators: managing applications should not require a bolt-on solution. Driving the service management interface into the operating system also brings significant benefits through encouraging evolution of core operating system functionality to support the service, and creating a truly common interface for all user software: operating system daemons and third party products alike.

The UNIX service management model has not evolved significantly beyond the traditional process model coupled loosely with service startup scripts, and is stretched in trying to meet modern system administration demands. The lack of functionality is costly for administrators, who must spend significant effort monitoring and managing systems that provide no fundamental abstractions to allow this.

SMF directly addresses all of these administration gaps and provides an integrated solution for service delivery and management on UNIX systems. SMF defines a fundamental service model which is used to control system startup, introduces a management interface for system and application services, and delivers diagnosis and restart capabilities for all services to maximize application availability.

Background and Related Work

The traditional System V and BSD style init.d/rc systems provide an extremely flexible mechanism for initiating arbitrary processes during system startup. For all of the flexibility provided, a number of deficiencies are apparent. We'll enumerate these deficiencies in terms of the System V implementation, but analogous problems exist in the BSD style system.

- No systematic way to list services. As the init.d system does not even encourage, much less enforce a 1:1 mapping of system services to init.d scripts, a simple ls of rc directories is never sufficient to get a full list of services provided by the system. The services required to start core operating system components are often the most opaque to the administrator.
- No persistent mechanism to indicate to the system whether a service should be running or not. Upgrade of software components often updates or even re-creates the corresponding rc script, overwriting administrative customizations such as attempts to disable the service.
- No formal dependency management interfaces. System components are usually very sensitive to ordering changes. The System V lexicographical ordering provides some latitude for the third-party service. Service developers can have their script run at a certain point during startup (before some services and after others) by giving the script an appropriate name. But, the lack of explicit dependency specification for a service makes modifications to the ordering of core system services fragile, and leads to unexpected behaviour.
- Incomplete manual restart capabilities. The individual init.d scripts are not necessarily stand-alone or idempotent. The administrator is not expected to execute all of the init.d scripts manually, so there is no simple way to restart many individual services without understanding the intricacies of their implementation. Closely cooperating services compound the problem; restart of one may require restart of its closely coupled dependent services. The init.d system provides no mechanism to determine service interdependencies, so significant administrative expertise is required to restart individual services without a reboot.
- Nonexistent automatic restart capabilities. The init.d system does not include automatic detection and restart of failed components. Detection and restart is left to orthogonal mechanisms such as inittab and inetd.conf.
- Lack of correlation between processes and services. Services are becoming increasingly sophisticated; the traditional UNIX model of a single daemon process as the entirety of a service is no longer adequate. Multiple processes are correlated into service boundaries by administrative expertise rather than being programmatically represented as a first-class object. The UNIX process model complicates the analysis – daemons are often reparented to init and lose their parent-child association in the process tree.
- Ticking configuration time bombs. There is no formal association between the current status of the system and the expected state when the system is restarted because starting a service has no impact on its settings on reboot. In addition, there is no systematic way to determine current versus configured state, so a reboot often leads to unpleasant surprises due to un-configured or mis-configured services.

For rigidly specified service models, such as inetd, the deficiencies can be more easily addressed. However, a complete system management architecture must provide a model sufficiently rich for all system services. SMF addresses each of these deficiencies for general software services.

Previous work in this area aims primarily to improve deficiencies of the init.d/rc system in system startup and manual service enable, disable and restart, up to and including a more formal statement of dependencies [1, 2, 3]. This work helps administrators gain better visibility into system service configuration and provide slight enhancements to administrative tools. However, if application availability is the crucial measurement, these sorts of enhancements only aid insofar as they slightly reduce the chances of administrative error.

Some recent efforts [4, 5] are more sophisticated and include simple service restart. The service can be restarted either when requests for that service come in, or restarted when the service exits unexpectedly. However, these efforts assume a single process is the primary provider of service, which is insufficient for a large class of business-critical applications.

Design Principles

Altering system service management is guaranteed to cause a change in administrative procedures. Changing something as fundamental as the init.d system requires administrators to spend time learning the new management paradigm. Thus, we took an ambitious approach so that the significant changes happened all together, rather than creating an incremental education cost over many software releases. The redesign of service management procedures must be comprehensive, and consider the full range of administrative procedures, software errors, and hardware errors.

The most fundamental requirement of this work is a simple, common model for all services. Services must be able to encompass nearly all capabilities of the system – simple daemons, network services, complex databases, and even non-process-based services such as dump device configuration. The management framework must

be flexible enough to allow common actions to be done easily and with no required administrative knowledge of the service implementation. A common model for all application and system services is required in order to thoroughly leverage administrator expertise; an administrator should be able to manage syslogd using the same basic commands as telnetd.

The service model must be extensible. New types of services, increased diagnosability, and expanded meta-configuration should be easy to support with no required enhancements to existing service descriptions.

Relationships between services are more complex than start time requirements. Many services are written carefully to be resilient to failure of their dependencies. For example, a well written networking application does not need to be restarted when a transient nameservice failure occurs. However, starting even a defensively-coded application without nameservice availability is usually futile. Thus, a rich dependency declaration is required to encompass both startup and failure scenarios.

Services then are restarted according to their dependencies following either hardware or software failure. A service restart must be attempted following any type of failure, such as a critical process exit, an uncorrectable memory error or other hardware error, a software coredump or premature exit, or an administrator accidentally killing the wrong process. Any dependents of that service must be restarted if they specify intolerance to dependency failures.

When a service cannot be automatically repaired after a fault, the system must provide significant aids to manual diagnosis, and trace failure back to the responsible component. When software failures can be traced back to the initial fault, this information as well as as much data as possible about the specific fault must be provided to the administrator so that he can spend his time repairing the fault rather than debugging an error in a dependent service. For example, if an application cannot start because the filesystem containing the application data was unable to mount, the system should point to the filesystem as the root cause of the fault, rather than incorrectly indicting the application.

The other critical component of a service management interface is its handling of service configuration data. We identified a set of meta-configuration, which is similar across a large set of services. For example, one critical pieces of meta-configuration captures whether a service is supposed to be running on a specific system. Common tools should be provided for meta-configuration, while still retaining the flexibility for services to provide their own complex application-specific configuration.

Meta-configuration must be available via a common command set and API across all services. This allows rapid development of higher-level administration tools which need no application knowledge to perform common tasks. Service-specific configuration can also be stored in a format accessible by this API, but the choice to transition to the new configuration store is up to the application developer depending on their compatibility requirements. Through the common commands, configuration rollback is provided. An administrator should be able to undo configuration or meta-configuration changes which rendered a service unstartable.

A few constraints were also required for a practical implementation in an operating system with strong compatibility requirements:

- No application binary changes required for basic participation in the framework.
- Compatibility for existing init.d services. The vast majority of services must continue to work, to account for lag time in new feature adoption by software developers.

Extending UNIX Processes to Software Services

SMF formalizes software ‘‘services’’, and introduces the tools to create, observe, manage and automatically restart them. A service is a long-lived software object (typically, a daemon) with a name, a clear error boundary, start and stop methods, and dependency relationships to other services on the system.

A service may be composed of zero, one, or many processes. A comprehensive software service model must allow all of these process models, and take each into account when defining failures. We’ll first consider a typical UNIX service which only has one process. These services are relatively straightforward to manage and monitor. The single process makes process-to-service mapping easy to understand, as no complex lineage from parents to children needs to be tracked. If the process exits, the service is no longer being provided and must be restarted. Still, without a new mechanism to track process faults, most monitoring implementations would require services forgo the traditional reparent-to-init step.

Even simple services can consist of multiple processes. For example, the Sendmail service usually includes two processes. Failure of either process should cause the service to be restarted. Traditional models for determining the relationship between these two processes fall down: they both re-parent themselves to init, breaking monitoring implementations which rely on the parent-child relationship. Aside from the process name, it is difficult to tell they’re both part of the same service.

Finally, we have a set of services which only appear transiently during the startup process. Usually, they exist to execute a command or small set of commands which change configuration state, such as informing the kernel about certain configuration parameters. It is important for SMF to understand a priori that a lack of long-running processes in these *transient* services does not constitute a fault.

In order to monitor the health for all three types of services, SMF introduces a kernel interface called a contract. A *process contract* allows a userland process to register an interest in a process and all of its children. The process receives reliable events when important changes occur, such as when all processes in the contract exit, a hardware error occurs in any of the processes, any processes in the contract coredump, or any process in the contract receives a fatal signal. All processes on the system must belong to a process contract, and all children of a process are part of the same contract until one creates a new contract.

Service *restarters* are responsible for managing software services. A restarter can use process contracts to receive and respond to fault events for a service. Usually, a restarter will write a process contract for each new service it creates. Thus, events for each service will be sent individually to the restarter, where it can make decisions about how to respond to contract events and administrative requests. Each process can fail separately, but it is the restarter's job to decide how a process failure should affect the service.

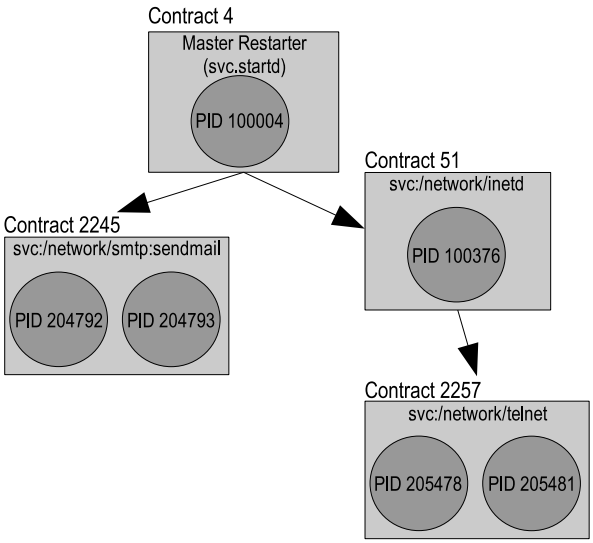

Figure 1: Contract model for software services.

A restarter determines how to interact with a service through its meta-configuration, which is stored in the SMF *repository*. The SMF repository stores persistent configuration and meta-configuration values as well as shorter-lived status information about each service. Each service has a common set of meta-configuration:

- The service name, in the form of a Fault Managed Resource Identifier (FMRI) is the unique identifier of the service on the system. The full FMRI for the sendmail service is: svc:/network/smtp:sendmail, but abbreviations like "sendmail" are available for interactive use.
- Service *dependencies* specify the relationships between services.
- *Method* specifications tell the restarter how to invoke the services, and may include invocations for start, stop, etc.
- The restarter defined by the service is responsible for starting, stopping, and restarting the service. Most services are managed by the default restarter, svc.startd. We've also implemented inetd as a service restarter.
- Services can optionally include localized, human readable descriptions and documentation references.

A service may also use the SMF repository to store simple configuration information. This allows application and system service developers to avoid maintaining configuration file parsers for small, simple configurations.

We developed a comprehensive state model for managing services, which are always in one of the following states: uninitialized, offline, online, degraded, disabled, and maintenance. These states have a consistent definition regardless of the service model.

State	Significance
Uninitialized	The initial state for all services. After evaluation by its restarter, the restarter will move the service to a maintenance, offline, or disabled state.
Disabled	The service has been disabled and is not running.
Offline	The service is enabled, but is not yet running. Often, a service will remain offline due to an unsatisfied dependency.
Online	The service is enabled, has started successfully and is currently running.
Degraded	The service is enabled, currently running but may be functioning at a limited capacity (e.g., reduced performance). The precise definition of degraded is service-specific.
Maintenance	The service is unavailable and cannot be automatically repaired by SMF. Administrative intervention is required to repair the service and clear the fault.

Table 1: State model for managing services.

Services transition between these states either because of administrative action (system startup, service enable, service disable), or service error (core dump, starting too rapidly, hardware fault). The service's ability to transition to a given state is always additionally influenced by the states of its dependencies.

To implement the service state model, SMF includes service restarters, which are themselves system

services (and hence subject to the state model). In our current work, a master restarter (svc.startd) and one delegated restarter (inetd) have been developed. While each of these are concerned with the restart of UNIX processes, the restarter architecture also allows implementation of non-process-based service models [6]. Each service restarter is responsible for defining appropriate transitions between the available SMF states.

Service Development

The meta-configuration and other simple service configuration are delivered onto the system by a small XML file, known as a service *manifest*. When the system starts up, the individual manifests are imported into the main SMF repository. The configuration of the system, including administrative customizations and run-time service data is stored in the repository and accessed by a common set of commands and APIs. The repository also provides the transactional semantics required for recovery on failure as well as the previous configuration snapshots which allow for configuration rollback.

In order to deliver an SMF-aware service, the administrator or software developer is required to create a service manifest. The manifest must include the meta-configuration for the service: a service name, the required methods, all dependency information, documentation references, and any other configuration to be stored in the repository.

SMF currently provides two restarters: svc.startd, the master restarter, and inetd, the inet service restarter. Both restarters require service definition through a manifest.

To provide a smooth upgrade path, service manifest creation can be an automatic step for inetd.conf services. We provide the inetconv utility to convert entries from inetd.conf to service manifests on operating system upgrade and allow administrators to convert entries subsequently added to the file. This is made easy by the well-specified service model for inetd services, and the fact that their dependencies are well known: inetd itself must be started, and rpc services require rpcbind.

Manifest creation for init.d scripts remains a manual process. The set of information required is well-defined, however two specific questions cannot be answered programmatically:

1) What are this service's dependencies? While we might be able to imagine a run-time checker or code analyzer for dependencies, these dependencies are often subtle and strongly tied to configuration. Automatic application dependency analysis would be a fascinating research topic of its own.

   The service author is usually best equipped to define the dependencies precisely. However, any consumer of the service can usually specify a dependency set that is sufficient for normal operation.

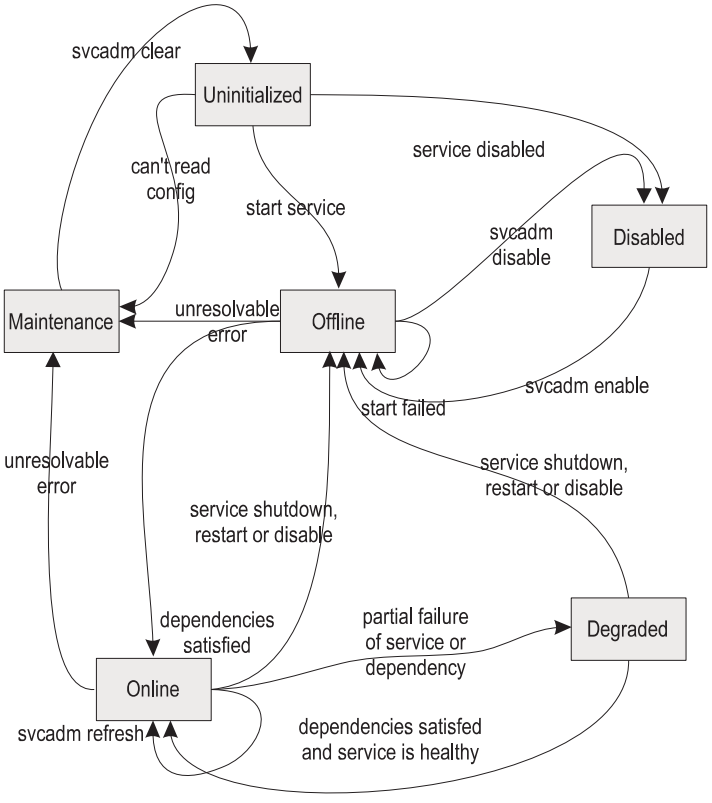

Figure 2: svc.startd state transitions.

2) What is this service's runtime behaviour? Does it have long-running processes that must be monitored, or is a lack of associated processes considered normal?

The service manifest author is required to specify dependencies in the manifest. Dependencies define service startup order as well as the restart relationships between services. A dependency is said to be satisfied if all conditions of its definition are met. There can be multiple dependencies for each service and each dependency may declare interest in multiple services. Two attributes of a dependency aside from the actual services in the dependency must be defined. The *grouping* specifies how the state of the services specified in the dependency are evaluated to determine whether a dependency is satisfied:

- *require_all*: all named services are running (online or degraded)
- *require_any*: at least one named service is running (online or degraded)
- *optional_all*: all named services must be either running (online or degraded), disabled, in maintenance, or not present on the system
- *exclude_all*: all named services must be either disabled, offline, in maintenance, or absent

The *restart_on* property determines which events should cause the service to be stopped:

- none: required only when the service is started

```
<service_bundle type='manifest' name='SUNWcsr:utmpd'>
<service
   name='system/utmp'
   type='service'
   version='1'>
   <create_default_instance enabled='true' />
   <single_instance/>
   <dependency
      name='milestone'
      grouping='require_all'
      restart_on='none'
      type='service'>
      <service_fmri value='svc:/milestone/sysconfig' />
   </dependency>
   <dependent
      name='utmpd_multi-user'
      grouping='optional_all'
      restart_on='none'>
      <service_fmri value='svc:/milestone/multi-user' />
   </dependent>
   <exec_method
      type='method'
      name='start'
      exec='/lib/svc/method/svc-utmpd'
      timeout_seconds='60' />
   <exec_method
      type='method'
      name='stop'
      exec=':kill'
      timeout_seconds='60' />
   <stability value='Unstable' />
   <template>
      <common_name>
         <loctext xml:lang='C'> utmpx monitoring
         </loctext>
      </common_name>
      <documentation>
         <manpage title='utmpd' section='1M' manpath='/usr/share/man' />
         <manpage title='utmpx' section='4' manpath='/usr/share/man' />
      </documentation>
   </template>
</service>
</service_bundle>
```

Figure 3: utmpd manifest.

- error: stop the service if the dependency fails due to hardware, software or administrative error
- restart: stop the service if the dependency stops for any reason
- refresh: stop the service if the dependency stops or is refreshed

Explicit dependency specification makes parallel service startup easy, regardless of whether the system is booting for the first time, or if a set of services had failed and are being restarted. Services are always started as soon as their dependencies are satisfied, so maximum parallelism for starting services is always achieved.

Unifying application deployment also provides a single location for access to advanced features. A service can be run with limited Privileges [11], or as an unprivileged or non-root user. Service management privileges may be given to authorized users without giving them full root access. A service may be bound to specific resource management limit and goal sets. All of this can be specified as service configuration with no code changes to the application itself (see Figure 4). Management of Solaris Zones [9] is simpler as SMF is available to each zone administrator.

Service Administration

The primary benefit a unified service management framework brings to system and service administration is a meaningful system-level view of all critical applications (see Figure 5). Services in an unexpected state are sorted at the bottom of the output for simple human consumption.

A primary goal of the SMF administrative model is to make common questions about services easy to answer and make common system administration tasks simple to perform. To evaluate the simplicity of the SMF model, the following provides examples of a few of these questions that were particularly difficult to answer in the init.d and inetd models prior to SMF.

What processes make up this service?

```
$ svcs -p sendmail
STATE   STIME   FMRI
online  May_06  svc:/network/smtp:sendmail
        May_06      9456 sendmail
        May_06      9458 sendmail
```

What's wrong with my system? (This will be covered in more detail in the next section.) See Figure 6.

What services does my service require in order to start? See Figure 7.

Which services won't be able to run if I disable this service? See Figure 8.

What services are available on this system? See Figure 9.

Messaging is also under the control of the administrator rather than the service. Messages previously emitted to console are stored in a per-service logfile, where they can be perused as part of post-mortem debugging or other diagnosis. They won't be lost to an insufficient terminal buffer size.

SMF defines a specific set of common administrative actions which can be applied to services:

```
# svccfg -s <svc> setprop start/user = astring: daemon
# svccfg -s <svc> setprop start/group = astring: daemon
# svcadm refresh <svc>
# svcadm restart <svc>
```

Figure 4: Configuring a service to run as ''daemon'' user and group.

```
$ svcs
STATE          STIME    FMRI
[...]
legacy_run     May_02   lrc:/etc/rc3_d/S81volmgt
legacy_run     May_02   lrc:/etc/rc3_d/S84appserv
legacy_run     May_02   lrc:/etc/rc3_d/S90samba
online         May_02   svc:/system/svc/restarter:default
online         May_02   svc:/network/pfil:default
online         May_02   svc:/network/loopback:default
online         May_02   svc:/system/filesystem/root:default
online         May_02   svc:/system/filesystem/usr:default
[...]
```

Figure 5: Abbreviated svcs output, including legacy services.

```
$ svcs -x
svc:/network/smtp:sendmail (sendmail SMTP mail transfer agent)
 State: maintenance since Tue May 10 18:35:41 2005
Reason: Method failed repeatedly.
   See: http://sun.com/msg/SMF-8000-8Q
   See: sendmail(1M)
   See: /var/svc/log/network-smtp:sendmail.log
Impact: This service is not running.
```

Figure 6: Checking the sendmail service.

- **enable** – Mark the service as enabled and start the service after all dependencies are satisfied.
- **disable** – Mark the service as disabled, stop the service, and do not allow it to start again.
- **refresh** – Reload service configuration and run the service's refresh method (if a refresh method is defined by the service).
- **restart** – Stop the service, then start it after its dependencies are satisfied.
- **clear** – Mark a service in the maintenance state as repaired, and if it is enabled allow it to start after all its dependencies are satisfied.

Administrators use svcadm to perform these administrative actions. Administrative intent is always preserved; disabling a service is guaranteed to persist across even patch and upgrade boundaries, where that was difficult in the past. The separation of administrative action and service state allows easier evaluation when system state doesn't match the administrative desire.

The use of a single API to take administrative action and change service configuration allows one point for security enforcement. Thus, SMF interoperates intimately with Role Based Access Control [11] mechanisms. It is easy to give a user just the ability to take action on a service (e.g., restart the service), but not change the service's configuration (see Figure 10). An administrator may also delegate authorizations with more granularity, giving privilege for an application

```
$ svcs -d sendmail
STATE  STIME  FMRI
online Aug_12 svc:/system/identity:domain
online Aug_12 svc:/system/filesystem/local:default
online Aug_12 svc:/network/service:default
online Aug_12 svc:/milestone/name-services:default
online Aug_12 svc:/system/filesystem/autofs:default
online Aug_12 svc:/system/system-log:default
```

Figure 7: Determining services needed by sendmail.

```
$ svcs -D system-log
STATE          STIME    FMRI
disabled       Aug_12   svc:/system/auditd:default
disabled       Aug_12   svc:/application/print/server:default
disabled       Aug_12   svc:/network/rarp:default
online         Aug_12   svc:/milestone/multi-user:default
online          5:55:34 svc:/network/smtp:sendmail
```

Figure 8: Determining services which require system-log's.

```
$ svcs -a
[...]
disabled       Aug_12   svc:/network/iscsi_initiator:default
disabled       Aug_12   svc:/system/metainit:default
disabled       Aug_12   svc:/network/ipfilter:default
disabled       Aug_12   svc:/network/rpc/nisplus:default
disabled       Aug_12   svc:/network/nis/server:default
disabled       Aug_12   svc:/network/ldap/client:default
[...]
```

Figure 9: Available service listing.

```
$ id
uid=37436(lianep) gid=10(staff)
$ grep lianep /etc/user_attr
lianep::::auths=solaris.smf.manage
$ svcs sendmail
STATE          STIME    FMRI
disabled       18:51:56 svc:/network/smtp:sendmail
$ svcadm enable sendmail
$ svcs sendmail
STATE          STIME    FMRI
online         18:52:43 svc:/network/smtp:sendmail
$ svcadm refresh sendmail
$ svcs sendmail
STATE          STIME    FMRI
online         18:52:55 svc:/network/smtp:sendmail
$ svccfg -s sendmail delpg autofs
svccfg: Permission denied.
```

Figure 10: Delegating service management authorizations.

administrator to only manage and change the configuration of a single service [10].

Finally, system security can be easily configured by creating a *profile*, which explicitly specifies which services are enabled and disabled on a system. To satisfy the common system configuration goal of no unencrypted network login services running, we provide the limited networking profile. Applying this profile explicitly disables all unencrypted network login services and enables other important services like ssh (see Figure 11).

Service Diagnosis and Self-Healing

svcs -x is a quantum leap forward in diagnosing problems with systems. It provides a very powerful paradigm. It describes what's going on in plain language with simplified output. It includes documentation references, as direct access to more information speeds the repair process. It points to an online knowledgebase; the website link included in the output contains the most up-to-date information on how to resolve the specific problem seen. Finally, svcs -x gives an assessment of the impact of each problems. If the specific issue effects no other services, that is stated explicitly. If other services are affected, svcs -xv will list them.

As the system (through dependency information) can determine the root cause of the problem, it can point the administrator directly to the component that must be repaired (see Figure 12).

In many cases, though, the administrator never needs to handle an error manually. Failure of a service due to hardware error, software bug, or administrative error can usually be resolved by restarting the service. This is handled automatically, with no administrative intervention. SMF also logs the error cause, to the extent it is known at the time of failure. This automatic recovery significantly reduces administrative costs.

To implement service restartability, SMF needed a way to detect when an error occurred in the service. Contracts provide a generic mechanism to express a relationship between a process and the kernel-managed resources it depends upon. The process contract allows

```
# svcs telnet
STATE          STIME    FMRI
online         17:49:15 svc:/network/telnet:default

# cd /var/svc/profile/
# cat generic_limited_net.xml
<service_bundle type='profile' name='generic_limited_net'
      xmlns:xi='http://www.w3.org/2003/XInclude' >
[...]
  <service name='network/ssh' version='1' type='service'>
    <instance name='default' enabled='true'/>
  </service>
[...]
  <service name='network/telnet' version='1' type='service'>
    <instance name='default' enabled='false'/>
  </service>
[...]
</service_bundle>

# svccfg apply generic_limited_net.xml
# svcs telnet
STATE          STIME    FMRI
disabled       17:49:40 svc:/network/telnet:default
```

Figure 11: Viewing and applying the limited network profile.

```
$ svcs -xv
svc:/system/filesystem/local:default (Local filesystem mounts)
 State: maintenance since Tue Sep 27 19:03:43 2005
Reason: Start method exited with $SMF_EXIT_ERR_FATAL.
   See: http://sun.com/msg/SMF-8000-KS
   See: /var/svc/log/system-filesystem-local:default.log
Impact: 23 dependent services are not running:
        svc:/system/sysidtool:net
        svc:/network/rpc/bind:default
        svc:/network/nfs/status:default
        svc:/network/nfs/nlockmgr:default
        svc:/network/nfs/client:default
        svc:/system/filesystem/autofs:default
        [...]
```

Figure 12: Service diagnosis of filesystem mount failure.

development of sophisticated restarters, which create a fault boundary around a set of processes, and receive and respond to events on processes within that boundary.

Implicit in service recovery is that the framework itself must also be fully restartable in the face of failures. An error of any user-land framework component all the way back to init can be caught, and the framework component itself will be restarted (see Figure 13). A transactional repository is required in order to implement the algorithms which recover from failure at any point.

Systems with sophisticated hardware error handling [7] make software recovery after error even more critical. Prior to SMF, the operating system developer when handling a hardware failure was given only a few unenviable options: kill the affected process and risk cascading failure, or restart the entire system. The operating system could not determine the broader effect of a faulted cell on a DIMM without inter-service dependency information. SMF manages error flow between services so that failures can be handled gracefully, by shutting down only affected processes and services which depend upon them.

Availability

The first version of the Service Management Facility is an integrated component of the Solaris 10 Operating System, released in January 2005. Themost recent copy of Solaris may be obtained free of charge at http://www.sun.com/software/solaris/ .

The Service Management Facility is also an integral part of OpenSolaris, which may be downloaded as source or binaries at http://opensolaris.org/os/ . The OpenSolaris SMF community contains information and discussion about service development and management in Solaris and OpenSolaris: http://opensolaris.org/os/community/smf/.

Experience

The first proof of concept and vetting of the design occurred when transitioning approximately 100 system services from their init.d script components into SMF. We wanted to confirm certain aspects of the SMF design, as well as take advantage of SMF benefits for Solaris service administration. The majority of services delivered as part of the Solaris operating system were migrated to SMF as part of its initial integration. We'll explore this case study here.

A small engineering team was responsible for creating nearly all of the 100 manifests for system services. A major lesson was that once a developer gains familiarity with the SMF model, manifest creation tends to take no more than a few hours per service, often including necessary testing. Experience with one manifest is directly leveraged for subsequent manifest creation. The fundamentals for basic service cooperation in SMF are relatively easy to grasp and do not seem to require significant training. The more sophisticated aspects of the service model tend to not be explored by the average service author, which we interpreted as success in our goal to make simple services easy to define.

As expected, the most challenging part of manifest creation was researching proper dependencies for each service. Sometimes the initial dependency analysis was incomplete – but, both point fixes for significant problems and longer term dependency additions to fill gaps were easy to perform.

We learned that someone familiar with the service implementation will be able to write a significantly better manifest than someone merely conversant in the service. A service manifest written by an end-user of the software is sufficient for that user's configuration, but not all potential uses and configuration of the service. Encouraging service authors to write

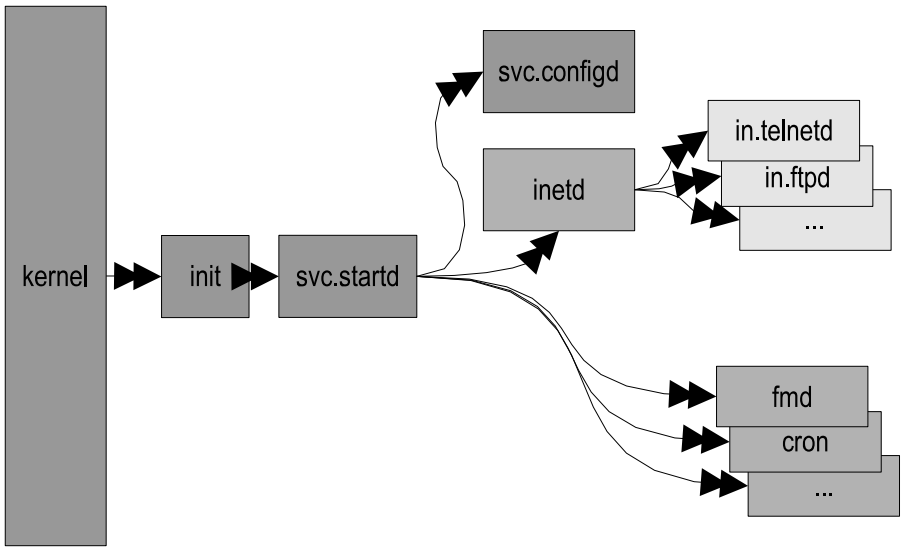

Figure 13: SMF framework restart relationships.

manifests for their software has significant value, even when a basic manifest has already been written.

Console interaction was more of a challenge than anticipated. When system startup was serial, console ownership was easy as it simply passed linearly from one script to another. SMF's parallel startup required tracking the console owner at all times, in case a system maintenance mode (also known as sulogin) was required to repair the system. The console ownership problem is a compelling reason for Solaris to provide access to the standard console login prompt as early in the boot sequence as possible. This convention is particularly helpful in a repair scenario because as much of the system as possible is available for use during the repair, rendering the repair environment less restrictive and more familiar.

Confirming the SMF design with critical system services started early in the boot sequence was likely a bit painful for the very early adopters. However, the lessons learned through actual service debugging informed significant usability enhancements which may not have been given the attention they deserved had SMF been an optional feature used only for a small subset of services.

Conclusions

A full-featured application and system service management infrastructure must reduce management complexity, increase application availability, and save administrators time. In order to provide all of these capabilities for a wide range of application models, the service must be elevated to a first-class administrative object which can be observed and managed. The Service Management Facility provides all of these benefits while requiring no changes to application binaries.

SMF reduces complexity by unifying and simplifying common administrative tasks across a broad set of applications. It increases application availability by detecting many critical errors and recovering from them automatically. Finally, it saves administrators time; when automatic recovery from failures is impossible, a complete management interface guides the administrator to the faulty component.

Author Biographies

Jonathan Adams joined Sun Microsystems, Inc. in Menlo Park four years ago, where he is a software developer in the Solaris Kernel Development group. His areas of expertise include memory allocation, inter-process communication, and debuggability. Reach him electronically at jonathan.adams@sun.com .

David Bustos graduated from the California Institute of Technology in 2002 with a BS in computer science. Since then he has worked in the Solaris operating system engineering group at Sun Microsystems, in Menlo Park, California.

Stephen Hahn is a Senior Staff Engineer in the Solaris Kernel Technologies group at Sun Microsystems. His recent work has been focused on service and resource management at the operating system level, particularly in building foundations for automated resource managers. His research interests are broad, and include describing meaningful application interdependencies, predictable systems behaviour, open source development processes, and implementing high performance sort algorithms. He received his Ph.D. in Theoretical Physics from Brown University, before joining Sun in 1997.

David Powell is a member of the Solaris kernel group at Sun Microsystems. In his six years at Sun, he has worked to improve the debuggability, availability, and approachability of Solaris, specifically focusing on inter-process communication and expressing dependencies between system resources and their consumers. He received his Sc.B. in Computer Science from Brown University

Liane Praza is a Staff Engineer in the Solaris Kernel Development group at Sun Microsystems. She's been at Sun since 1997, with areas of of expertise including the Solaris administration model, service and resource management, self-healing services, and clustered devices and filesystems. She received her B.S. in Computer Science from Purdue University. Reach her electronically at liane.praza@sun.com .

Acknowledgments

The SMF project was the work of a larger group of people than this paper represents, and we are grateful to everyone who worked to make this project possible. Dan Price, Dave Linder, and our USENIX shepherd, Tom Limoncelli, provided invaluable feedback and encouragement for this paper.

References

[1] Mewburn, Luke, "The Design and Implementation of the NetBSD rc.d system," *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[2] http://www.fastcoder.net/~thumper/software/sysadmin/chkconfig/ .

[3] Gooch, Richard, "Advanced Boot Scripts," *Proceedings of the Ottawa Linux Symposium*, June, 2002.

[4] http://developer.apple.com/documentation/MacOSX/Conceptual/BPSystemStartup/Articles/LaunchOnDemandDaemons.html .

[5] Bernstein, D. J., *Daemontools*, http://cr.yp.to/daemontools.html .

[6] Skinner, Glenn, et al., "A Service Management Facility for the Java Platform," *Proceedings of the 2005 IEEE Services Computing Conference*, 2005.

[7] Shapiro, Michael W., "Self-Healing in Modern Operating Systems," *ACM Queue*, Vol. 2, Num. 8, 2004.

[8] Candea, George, et al., "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel," *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001.

[9] Price, Daniel, et al., "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," *Proceedings of the 18th Large Installation System Administration Conferences (LISA, '04)*, 2004.

[10] Brunette, Glenn, *Restricting Service Adminisration in the Solaris 10 Operating System*, http://www.sun.com/blueprints/0605/819-2887.pdf.

[11] Sun Microsystems, Inc., *System Administration Guide: Security Services*, http://docs.sun.com/app/docs/doc/816-4557.