SERVICE MANAGEMENT FACILITY (SMF) IN THE SOLARIS 10 OS

Rob Romack, PTS (Mid-Range Server Group)

Sun BluePrints™ OnLine — February 2006

Part No 819-5150-10
Revision 1.0, 2/1/06
Edition: February 2006

Please
Recycle

Table of Contents

Service Management Facility (SMF) in the Solaris 10 OS

Introduction

A significant challenge in today's data centers is the demand for increased service levels in environments that feature increasing complexity. The Solaris™ 10 Operating System (OS) introduces a new foundation that improves service levels by detecting and correcting component failures while simplifying systems management. This foundation — known as Predictive Self-Healing — includes new technologies that Sun has incorporated into its hardware and software products to maximize availability in the event of system faults. Overall, Predictive Self-Healing simplifies system administration and helps to contribute to a lower total cost of ownership (TCO) in the data center.

A key component of Predictive Self-Healing is the new Service Management Facility (SMF) in the Solaris 10 OS. SMF is designed to simplify the management of system and application services. It delivers new and improved ways to control services, and tries to restart failed services automatically. In addition, SMF allows administrators to define the relationships between services. It is now possible to define a service that is dependent on other services — a dependent service will not run unless the other services that it requires are already running. Through a set of new administrative interfaces, SMF allows services to be easily and consistently configured, enabled, and controlled, at the same time providing better visibility of errors and improved debugging capabilities to resolve service-related problems quickly when they occur.

About This Document

This BluePrint Article is intended for system administrators. It introduces the functionality provided by the Service Management Facility (SMF) and demonstrates the use of new SMF administrative commands. It assumes that the reader has a reasonable level of knowledge of the Solaris OS (in particular, of OS versions prior to Solaris 10), or of other UNIX systems in general. The article makes the assumption that the reader is *not* already familiar with SMF or other specifics of the Solaris 10 OS.

This BluePrint Article addresses the following topics:

- Features and benefits of SMF
- The SMF repository, SMF services, and service instances
- Components of a service, including SMF manifests and service identifiers
- How to administer SMF services, including SMF commands and examples of how to perform common SMF administrative tasks
- How to administer network services in SMF using the `inetadm` interface
- An example of how to create a new SMF service
- Service dependencies and examples of how dependencies are defined
- SMF initialization at boot time
- Milestone services and how to associate services with milestones
- Dependency cycles and the process of resolving them

In this article, the `Courier` font is used to represent command line entries, file names, or excerpts from system files. When examples of a command are given, the use of **`bold Courier`** type represents what is typed by the user, while `regular-face Courier` type indicates system prompts or system output.

An Introduction to SMF Services

In the most general sense, SMF services provide capabilities to applications and other services, both local and remote. In the Solaris 10 OS, most system services — such as network services (`ftp`, `telnet`, `rlogin`, etc.), file system services, security services, device services, print services, `cron` services, and so forth — are implemented as SMF services. Besides typical UNIX system services, the SMF management framework is designed to support third-party software application services, including web services and database services (such as starting an Oracle database daemon). In addition to representing running daemons, SMF services can also represent the configuration of a subsystem, the software state of a device, or a set of other services.

Features and Benefits of SMF

SMF offers many advantages, including:

- Simplified service administration. Services are objects that can be viewed and easily managed with a few simple administrative commands.
- Automated restart of failed services. SMF monitors service processes, and can proactively restart a service when it detects an administrative error, hardware fault, or service death.
- Persistent service configuration. Service definitions and configurations persist across reboots, even after installing OS upgrades or patches.
- Explicit dependencies. Relationships are defined between services to reflect that some services rely on the availability of other services.
- Easier debugging. Individual service log files make it easier to determine why a service isn't running.
- Faster boot/shutdown processes. SMF parallelizes the start/stop of services when possible.
- Delegated service administration. Administrators can securely delegate service-related tasks to non-root users, including the ability to configure, start, stop, or restart services.

In previous versions of the Solaris OS, `init` executed a series of rc scripts, which ran sequentially to start all system services. With SMF and the Solaris 10 OS, most system services are no longer started from rc scripts. Legacy rc scripts will continue to work, but moving customized rc scripts to SMF is strongly recommended — SMF-managed services are easier to administer and control, and debugging service-related problems is much easier.

In earlier versions of the Solaris OS, there was little consistency in how services were managed (especially in the process of enabling and disabling them). In the past, to disable a service or daemon, administrators often renamed a particular rc script to something that the system would ignore upon reboot. A common problem with this was patch installation or upgrades, which might restore the renamed files. SMF provides a more consistent and persistent means of managing services — for system and application services alike.

SMF Repository

At the core of SMF is the configuration repository, which stores service configuration information in local memory and local files. The repository provides a persistent way to enable or disable services, a consistent view of service state, and a unified interface to get and set service configuration properties. The repository provides a snapshot of each service's configuration at the time that the service successfully starts, which allows the service to be easily restored to a known good configuration.

SMF Restarters

In the Solaris 10 OS, the `init` process starts up the SMF master restarter daemon `svc.startd`. (a *restarter* is simply a program that restarts a service). The `svc.startd` daemon queries the SMF repository to locate other system services, and starts them according to their dependencies, in parallel whenever possible. This is in contrast to earlier versions of the Solaris OS and other UNIX operating systems that use rc scripts to start services in a serial fashion. As the SMF master restarter daemon initializes services, it enforces defined dependency relationships between services, starting only those services where dependency requirements have been met. (Dependencies are discussed later in this article — see "Service Dependencies" on page 18.)

Each service specifies a restarter which is used to initialize the service. If a service has not specified a restarter, then the default restarter `svc.startd` is used. If another restarter is specified, it is called a "delegated restarter". For many network services (such as `rlogin`, `ftp`, etc.), the Solaris OS defines `inetd` as the service restarter. The delegated restarter `inetd` performs some common actions (such as port binding) on behalf of the services it manages.

In addition to starting services, restarters keep track of service failures and dependency events. This allows a restarter to automatically restart a service when it detects a service has failed or one of its required dependencies is no longer available. In this way, restarters help to simplify and automate some service-related tasks.

SMF Service Instances

It is not uncommon for a system to run multiple copies of the same service, usually with slightly different configurations. For example, a web server is a service, but a specific web server daemon configured to listen on port 80 is an instance of that service. To facilitate configuration sharing, SMF extracts configuration properties for each service from *service instances*, which are represented in the repository by service instance objects. Service instance objects are children of service objects, and may contain configuration properties that are shared between the instances. If an instance does not explicitly override a configuration property, then it inherits property values from the parent service object.

The following example clarifies the distinction between a service and a service instance. The remote login server daemon, `in.rlogind`, has three distinct modes. By offering each mode as a separate instance of the same service, SMF can use a single, common configuration for all three, while allowing the administrator to enable or disable each mode independently. In the Solaris 10 OS, the service is named `network/login` and the service instances are:

- Instance 1: `svc:/network/login:rlogin` (rlogin)
- Instance 2: `svc:/network/login:klogin` (rlogin with Kerberos)
- Instance 3: `svc:/network/login:eklogin` (rlogin with Kerberos and encryption)

Components of a SMF Service

A SMF service may have some or all of the following entities associated with it:

- A SMF manifest that defines the default set of service properties
- One or more methods that define how the service's restarter interacts with the service
- One or more executables (or daemons) called by the methods to implement the service
- A log file that records the output of the service
- A Fault Management Resource Identifier (FMRI) used to identify a specific service instance

Minimally, a service can be defined using a set of methods and an FMRI, although some services may use all of the entities listed above. To illustrate SMF concepts and terminology, Figure 1 shows the files, executables, and identifiers associated with the `cron` service.

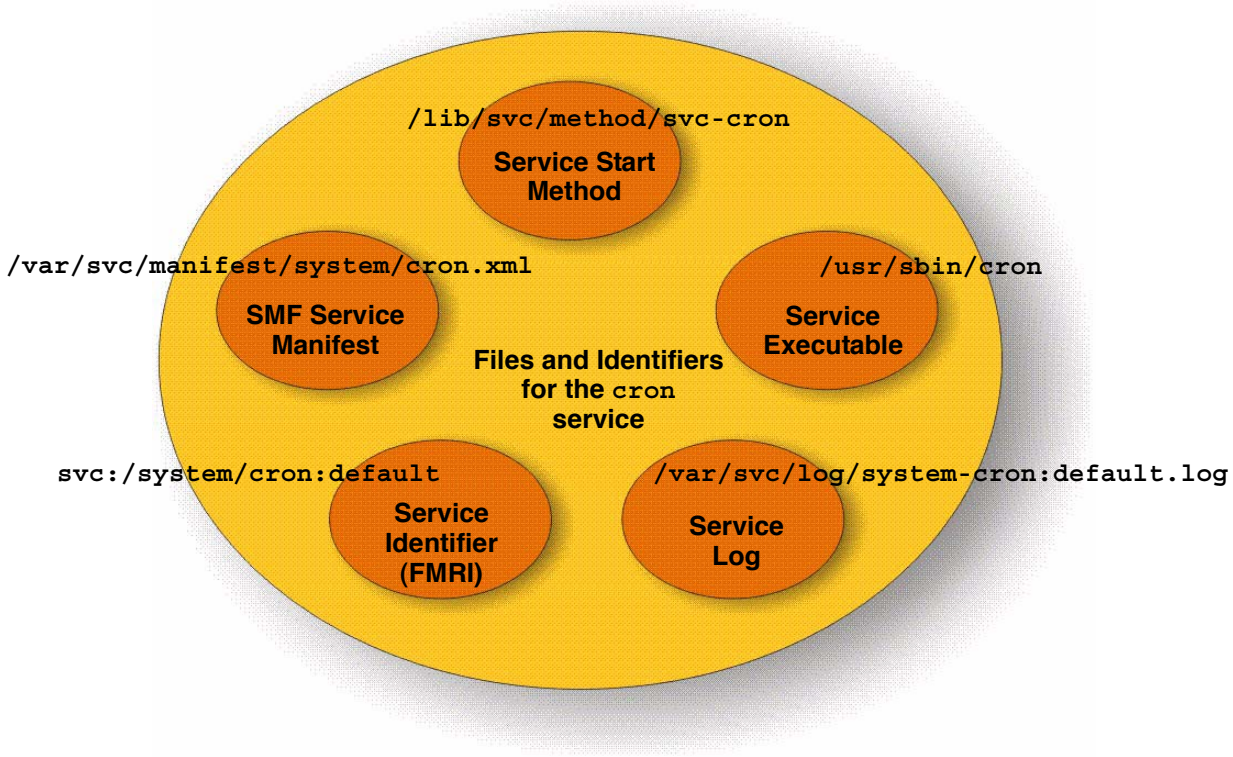

Figure 1. Files and identifiers related to the `cron` service

SMF Manifests

A SMF manifest is an XML file that contains a complete set of properties associated with a service or a service instance. System manifest files are stored in `/var/svc/manifest`, although manifests may exist elsewhere. (Note that only manifests in `/var/svc/manifest` are imported during boot.) SMF manifest files are used to create configurations within the repository, which then maintains the authoritative source of configuration information for all service instances. To import configuration properties from a new service's manifest into the repository, an administrator can either run `svccfg import` or allow the service to import the information during a system boot. (See the `service_bundle(4)`, `svccfg(1M)`, or `smf_bootstrap(5)` man pages.)

Note – It is *not* supported to change the properties of system services by directly modifying SMF manifest files provided in the Solaris 10 OS. If Solaris OS or ISV-delivered manifests are modified directly, customizations may not be preserved across software upgrades. The recommended way to change service configuration properties stored in the repository is with the `svccfg` command.

Service Methods

Methods are used by a restarter to interact with the service, and typically may be an executable, a shell script, or a keyword (such as "`:kill`"). For services managed by the master restarter `svc.startd`, methods are often shell scripts similar to traditional rc scripts in previous Solaris OS versions. The restarter `svc.startd` uses a "start method" to initialize a service, a "stop method" to halt it, and optionally a "refresh method" to reread its configuration. Methods for many system services reside in the `/lib/svc/method` directory. (For more information, see `smf_method(5)` or `svc.startd(1M)` man pages.)

Service Executable

In some cases, a start method invokes a service executable, which is responsible for providing the capabilities of the service. The daemon `/usr/sbin/cron`, for example, is the executable for the `cron` service (refer back to Figure 1). A service executable may also be invoked directly as a method.

Service Log Files

A service's restarter may specify a log file to capture information about the service. (For example, `svc.startd` logs actions about specific services to each service's log file in `/var/svc/log`.) Log files in `/var/svc/log` differ from log file(s) associated with an executable that may be invoked as part of the service. For example, the log file for the `cron` service is `/var/svc/log/system-cron:default.log`, whereas `/var/cron/log` contains the output of the daemon `/usr/sbin/cron`.

Service Identifiers — FMRIs

Each service instance is named with a Fault Management Resource Identifier (FMRI), which is used with SMF administrative commands to specify the service instance to be acted upon. An FMRI usually contains three parts separated by colons:

```
# svc:/network/login:rlogin
```

The first part of the FMRI — the `svc` prefix — indicates that the FMRI describes a Solaris SMF service. The second part gives the name of the service, in this case, `network/login`. The third part identifies the service instance (in this example, `rlogin`, which is an instance for the remote login service).

Equivalent formats for an FMRI include:

```
svc://localhost/system/system-log:default
svc:/system/system-log:default
system/system-log:default
```

The full FMRI describes a service uniquely (and should be used in shell scripts), but abbreviated forms are easier to type and work as long as there is only one instance and SMF would infer that instance. The following short-form could be used, for example, and SMF would infer the instance to be used:

```
system-log
```

The “default” name is used for the default instance of a service:

```
svc:/network/rpc/bind:default
```

In this case “default” is the default instance of `svc:/network/rpc/bind`.

See the SMF command man pages, such as `svcadm(1M)` or `svcs(1)`, for instructions about which FMRI formats are appropriate.

Administering SMF Services

Table 1 contains a summary of commands that are frequently used to administer SMF services. For common service-related tasks, Table 2 compares commands typically used in earlier OS versions to comparable SMF commands.

Table 1. SMF Management Tools

Command	Description	Examples

`svcs`	Displays information about service instances	`# svcs -a`
		Lists all services, enabled or not.
		`# svcs -p [FMRI]`
		Lists any processes associated with each service instance.
		`# svcs -l FMRI`
		Displays details about a particular service.
		`# svcs -x`
		Displays all broken services and gives a reason as to why SMF believes the service is broken. This command is a powerful troubleshooting tool, and can also be run with a “verbose” (-v) option.

`svcadm`	Issues requests for actions on executing services, including enabling, disabling, and restarting service instances	`# svcadm enable foo`
		`# svcadm refresh print/server`
		Enables and refreshes a service instance. (Note that it is necessary to refresh a service after any configuration changes via `svccfg`.)
		`# svcadm disable -t cron`
		Disables the `cron` service temporarily (until the next reboot). To disable a service persistently across reboots, do not use the “-t” option.

`svccfg`	Displays and manipulates the contents of the SMF repository	`# svccfg import /var/svc/manifest/system/foo.xml`
		Imports a service manifest into SMF.
		`# svccfg delete FMRI`
		Deletes (removes) the service definition from the SMF repository.
		`# svccfg`
		With no options, enters the `svccfg` interactive shell. In the interactive shell, the “help” subcommand lists other available subcommands.

`svcprop`	Retrieves property values from the SMF repository (with an output format appropriate for use in shell scripts)	`# svcprop -p propertygroup/property FMRI`
		Retrieves property values for the specified service instance.
		`# svcprop -p start/exec system/cron`
		For the `cron` service, retrieves the start method property.

`inetadm`	Provides the ability to observe and configure network services controlled by `inetd`	`# inetadm -p` Display global defaults for all `inetd` services. See also “Administering Network Services with inetadm” on page 11.

Table 2. A Comparison of Common Tasks

Task	Old Procedure	Comparable SMF Procedure
Enabling and disabling services	To disable system services like `cron`: **`rm /etc/rc2.d/S75cron`** (Repeat after every `cron` patch application and system upgrade.)	To disable system services like `cron`: **`svcadm disable cron`**
	Later to enable `cron`, reinstall: **`/etc/rc2.d/S75cron`**	Later to enable system services like `cron`: **`svcadm enable cron`**
	To enable inetd services (like `finger`), edit `/etc/inet/inetd.conf` and uncomment the service to be enabled. Then issue: **`pkill -HUP inetd`**	To enable `inetd` services (like `finger`:) **`svcadm enable finger`**
Stopping services	**`/etc/init.d/sshd stop`**	**`svcadm disable -t ssh`** (The "-t" indicates that the requested action should be temporary until the next reboot.)
Starting services	**`/etc/init.d/sshd start`**	**`svcadm enable -t ssh`**
Restarting services	**`/etc/init.d/sshd stop;`** **`/etc/init.d/sshd start`**	**`svcadm restart ssh`**
Refreshing the service configuration	**`kill -HUP `cat /var/run/sshd.pid``**	**`svcadm refresh ssh`**

In the comparable SMF commands, the last argument to `svcadm` is the service's FMRI. Note that the command `svcadm` can only be used for SMF services (services started and stopped in rc scripts can still be started and stopped as before, but cannot be managed using SMF commands).

Service States

At any point in time, a service instance may have one of the following states:

- uninitialized: This state is the initial state for all services before their configuration has been read or before their restarter has started.
- disabled: The service instance is not enabled and is not running.
- offline: The service instance is enabled, but cannot be started until its dependencies are met.
- online: The service instance is enabled and has successfully started.
- degraded: The service instance is enabled and running, but at less than full capacity.
- maintenance: The service instance has encountered an error that must be resolved by the administrator.
- legacy_run: This state is used only for legacy services. Legacy services cannot be controlled by SMF commands. (See the Solaris 10 System Administration Guide: Basic Administration (817-1985-xx), Chapter 14, “Managing Services".)

The `svcs` command displays the state, state time, and FMRI of service instances by default (use “-o” for alternate display options).

An Example of Using SMF Administrative Commands

A good way to learn about the SMF administrative commands is through an example. To illustrate the use of some commonly used commands, the example given here breaks the print service and then fixes it. Troubleshooting most service-related issues involves many of the same steps used to identify and resolve the problem in this example.

The example includes the following steps:

1. Modifying the configuration for the service (in which an error is made)
2. Restarting it (which causes it to fail)
3. Viewing the log file (which points to the error)
4. Modifying the configuration again (to repair the service configuration)
5. Restarting it (which is successful)

To modify the configuration of the print service, use `svccfg` in interactive mode to list the available services (for brevity, only an excerpt of the full `svccfg` output is given below):

```
# svccfg
svc:> list
system/console-login
milestone/devices
system/device/local
...
network/initial
network/loopback
network/physical
system/svc/restarter
system/filesystem/root
...
application/print/server
```

Select the service to be modified (`print/server`), and list the properties specifically for that service:

```
svc:> select print/server
svc:/application/print/server> listprop
start/exec astring  "/lib/svc/method/print-svc start"
...
```

The output shows that the start method (the start-up script) for the print service is `/lib/svc/method/print-svc`.

Modify the name of the start method using the `setprop` command. Change the name of the method to a non-existent file, which will subsequently cause an error when the print service is restarted. After making the change, refresh the configuration so that the change to the service's properties takes effect and then restart the service:

```
svc:/application/print/server> setprop
     Usage:  setprop pg/name = [type:] value
svc:/application/print/server> setprop start/exec = astring: "/lib/svc/method/print-svc.1
start"
svc:/application/print/server> quit      (exit the svccfg interactive session)
#
# svcadm refresh print/server
# svcadm restart print/server
```

Since the start method for the service (`lib/svc/method/print-svc.1)` does not exist, the service fails when it attempts to start.

To troubleshoot SMF service problems, begin by running `svcs -xv`. Many times, this simple command can point to what is broken and suggest a possible cause:

```
# svcs -xv
svc:/application/print/server:default (LP print server)
 State: maintenance since Sat Apr 30 09:38:25 2005
Reason: Start method failed repeatedly, last exited with status 1.
   See: http://sun.com/msg/SMF-8000-KS
   See: man -M /usr/share/man -s 1M lpsched
   See: /var/svc/log/application-print-server:default.log
Impact: 1 dependent service is not running:
        svc:/application/print/rfc1179:default
```

The output indicates a problem with the service's start method (it failed repeatedly) and suggests looking at the log of the broken service — `/var/svc/log/application-print-server:default.log`. The log file (shown below) indicates the source of the problem — the start method `/lib/svc/method/print-svc.1` was not found:

```
# tail /var/svc/log/application-print-server:default.log
...
[ Feb  6 11:27:52 Method "start" exited with status 1 ]
[ Feb  6 11:27:53 Executing start method ("/lib/svc/method/print-svc.1 start") ]
/sbin/sh: /lib/svc/method/print-svc.1: not found
```

Now that the problem is clearly identified, it can be fixed using the `svccfg` command (this time via the command line mode rather than the interactive mode). Once the start method is correctly defined for the service, the configuration can be refreshed and the service restarted:

```
# svccfg -s print/server setprop start/exec = astring:'"/lib/svc/method/print-svc start"'
# svcadm refresh print/server
# svcadm restart print/server
```

However, there is still a problem — the service is stuck in the maintenance state:

```
# svcs -p print/server
STATE          STIME    FMRI
maintenance   20:35:35  svc:/application/print/server:default
```

A service in the maintenance state requires administrative intervention. To remove a service from the maintenance state, an administrator needs to explicitly “clear” the service to signal that all necessary repairs are completed before the service can be restarted. After a service has been repaired, it must be cleared before it will go online. If it is not cleared, it will remain in the maintenance state. Use `svcadm` to clear the service as follows:

```
# svcadm clear print/server
# svcs -p print/server
STATE          STIME    FMRI
online         20:41:20 svc:/application/print/server:default
               20:41:20     1480 lpsched
```

Now the print service is online and ready to accept service requests.

Administering Network Services with inetadm

The management of built-in network services in the Solaris 10 OS is now handled through SMF. The Internet services daemon, `inetd(1M)`, has been rewritten as a part of SMF, and configuration data for network services is now stored in the SMF repository rather than in `/etc/inet/inetd.conf`. This allows the SMF tools to be used to control and observe `inetd`-based network services.

Any records remaining in `/etc/inet/inetd.conf` after an OS upgrade to the Solaris 10 OS, or later created by installing additional software, must be converted to SMF and imported into the SMF repository using `inetconv(1M)` — otherwise the service will not be available. To provide compatibility for services which have not yet been converted to SMF, entries can be added to `inetd.conf` using traditional syntax, and the utility `inetconv` will convert the new services to SMF. The command `inetconv` should always be run after changing `/etc/inet/inetd.conf`. Note that `inetconv` is run automatically during the first reboot after an OS installation or upgrade. If `inetd.conf` has been modified since `inetconv` was run, then `inetd` sends reminder messages to `syslog`.

After `inetd.conf` has converted a service, `inetadm` can be used to change properties of the converted service. The SMF utilities (such as `svcs`, `svcadm`, etc.) can also be used to observe the service and perform common actions.

The following commands show how to use `inetadm` to display and then set the tcp_wrappers attribute for the `ftp` service:

```
# inetadm -l ftp
SCOPE    NAME=VALUE
         name="ftp"
         endpoint_type="stream"
         proto="tcp6"
         isrpc=FALSE
         wait=FALSE
         exec="/usr/sbin/in.ftpd -a"
         user="root"
default  bind_addr=""
default  bind_fail_max=-1
default  bind_fail_interval=-1
default  max_con_rate=-1
default  max_copies=-1
default  con_rate_offline=-1
default  failrate_cnt=40
default  failrate_interval=60
default  inherit_env=TRUE
default  tcp_trace=FALSE
default  tcp_wrappers=FALSE      (tcp_wrappers attribute is disabled here)

# inetadm -m ftp tcp_wrappers=TRUE
# inetadm -l ftp
SCOPE    NAME=VALUE
         name="ftp"
         endpoint_type="stream"
         proto="tcp6"
         isrpc=FALSE
         wait=FALSE
         exec="/usr/sbin/in.ftpd -a"
         user="root"
default  bind_addr=""
default  bind_fail_max=-1
default  bind_fail_interval=-1
default  max_con_rate=-1
default  max_copies=-1
default  con_rate_offline=-1
default  failrate_cnt=40
default  failrate_interval=60
default  inherit_env=TRUE
default  tcp_trace=FALSE
         tcp_wrappers=TRUE      (tcp_wrappers attribute is now set)
```

The following commands show how the `svcadm` and `svcs` commands can be used to control and monitor the `ftp` service:

```
# svcadm disable ftp
# svcs ftp
STATE          STIME    FMRI
disabled        9:09:26 svc:/network/ftp:default

# svcadm enable ftp
# svcs ftp
STATE          STIME    FMRI
online          9:09:47 svc:/network/ftp:default
```

inetd as a Restarter

For many network services, `inetd` is the specified restarter, and it sometimes called a “delegated restarter” (i.e., it manages the starting and stopping of services in lieu of the default restarter `svc.startd`). For example, in the SMF repository for the `ftp` service, the service `inetd` is listed as the restarter. (By default, the manifest for the `ftp` service, `/var/svc/manifest/network/ftp.xml`, specifies `svc:/network/inetd:default` as the restarter.)

At boot, the master restarter `svc.startd` starts the service `inetd`, which in turn listens for requests for network services such as `ftp`. When an incoming `ftp` request occurs, `inetd` determines that the request is for the `ftp` service (`network/ftp`), and invokes the appropriate start method (`/usr/sbin/in.ftpd -a`). Figure 2 shows the interaction of `inetd` and the `ftp` service.

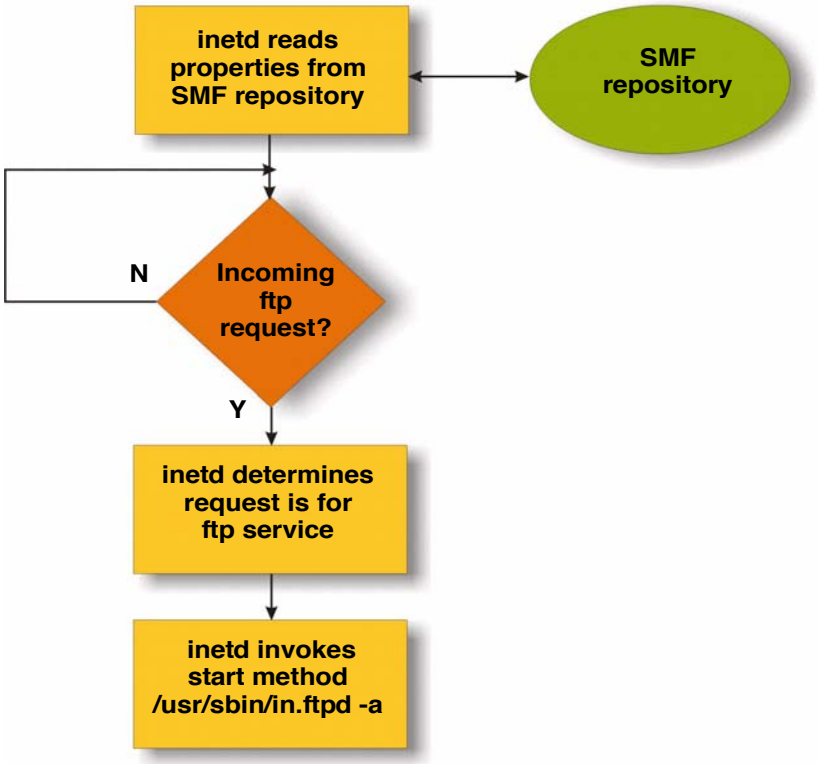

Figure 2. inetd-based services (such as ftp) are managed within SMF

Creating a New SMF Service

The easiest way to add a new service to SMF is to write an XML manifest that describes the service, each instance, and its properties. The manifest defines:

- Service methods. The manifest specifies a set of methods to start, stop, and, optionally, refresh the service. (Once service properties are defined in the SMF repository, the `svccfg` command can be used to list and modify the service methods.)
- Dependencies. Service instances may have dependencies on other services or files, and those dependencies govern when the service is started and automatically stopped. When the dependencies for an enabled service are not satisfied, the service is kept in the offline state. When all of the service's dependencies are satisfied, then the service is started and transitioned to the online state.
- Restarter. Each service is managed by a single restarter. The master restarter, `svc.startd`, manages states for many service instances and their dependencies. For some services, a delegated restarter (such as `inetd` for network services) is specified.

Constructing a New SMF Service — An Example

A step-by-step example will help to explain how to create a new service within SMF. Constructing the `foo` service requires the following components:

- The daemon `/opt/SUNWsmftest/bin/foo`. This is a program with no controlling tty that constantly runs and looks for service requests. In this example, the `foo` program is simply a placeholder and does nothing useful.
- The method `/opt/SUNWsmftest/lib/svc-foo.sh`. This is a shell script that starts the daemon.
- The XML manifest file `/var/svc/manifest/system/foo.xml`. The manifest describes the service and its properties to the SMF repository.

The method (`/opt/SUNWsmftest/lib/svc-foo.sh`) is the following shell script:

```
#!/sbin/sh
. /lib/svc/share/smf_include.sh

   if [ -x /opt/SUNWsmftest/bin/foo ]; then
      /opt/SUNWsmftest/bin/foo
   else
      echo "/opt/SUNWsmftest/bin/foo is missing or not executable."
      exit $SMF_EXIT_ERR_CONFIG
   fi

   exit $SMF_EXIT_OK
```

For the sake of simplicity, this example uses the XML manifest of the existing `cron` service — the `/var/svc/manifest/system/foo.xml` file is actually a copy of the `cron` manifest with a few simple modifications. It is beyond the scope of this article to extensively discuss the writing of SMF manifest files using XML, but for more information, see the article “Predictive Self-Healing: Solaris Service Management Facility — Service Developer Introduction” on the BigAdmin System Administration Portal (http://www.sun.com/bigadmin/content/selfheal/sdev_intro.html). In addition, the syntax specification for SMF manifests is available in `/usr/share/lib/xml/dtd/service_bundle.dtd.1`, and may be helpful.

Note – It is not supported to edit any Solaris 10 OS XML manifest files shipped with the OS to modify built-in system services. Changes to existing system services should be made using `svccfg`. In this example, a *copy* of an XML manifest is edited to create a new service.

Here is the contents of the XML manifest file `foo.xml`:

```
<?xml version="1.0"?>
<!DOCTYPE service_bundle SYSTEM "/usr/share/lib/xml/dtd/service_bundle.dtd.1">
<service_bundle type='manifest' name='SUNWcsu:foo'>

<service
    name='system/foo'
    type='service'
    version='1'>

    <single_instance />

<!-- foo_opt says that foo depends on local filesystem /opt -->
    <dependency
        name='foo_opt'
        type='service'
        grouping='require_all'
        restart_on='none'>
        <service_fmri value='svc:/system/filesystem/local' />
    </dependency>

<!-- foo_cron says that foo depends on svc:/system/cron -->
    <dependency
        name='foo_cron'
        type='service'
        grouping='require_all'
        restart_on='refresh'>
        <service_fmri value='svc:/system/cron' />
    </dependency>

    <exec_method
        type='method'
        name='start'
        exec='/opt/SUNWsmftest/lib/svc-foo.sh'
        timeout_seconds='60'>
        <method_context>
            <method_credential user='root' group='root' />
        </method_context>
    </exec_method>

    <exec_method
        type='method'
        name='stop'
        exec=':kill'
        timeout_seconds='60'>
    </exec_method>

    <instance name='default_foo' enabled='false' />

    <template>
      <common_name>
            <loctext xml:lang='C'>
            A simple example created by the author
            </loctext>
        </common_name>
    </template>

</service>
</service_bundle>
```

In the first part of the XML manifest above, notice the `name` field, which contains the FMRI service identifier `system/foo`. The two `dependency` blocks define dependencies called `foo_cron` (which states that `foo` depends on the service `system/cron)` and `foo_opt` (which states that `foo` depends on the filesytem `/opt`). The first `exec_method` block references the start method, `/opt/SUNWsmftest/lib/svc-foo.sh`, and specifies the credentials with which the start method will be executed (`user` and `group` are "`root`"). The second `exec_method` block includes the keyword "`:kill`" for the stop method, although a separate stop method shell script or other executable could have been defined instead. (The syntax "`:kill -9`" could also have been used; however this is a more drastic approach because it kills the process without allowing it to gracefully exit.)

In the commands below, `svccfg import` reads in the XML manifest and adds the `foo` service to the SMF repository:

```
# svccfg import /var/svc/manifest/system/foo.xml
# svcs foo
STATE      STIME    FMRI
disabled   10:56:21 svc:/system/foo:default_foo
```

At this point, the new `foo` service is defined within SMF, but the instance is in the disabled state. (This is preferred so that an administrator can precisely control the start of a new service, rather than having it start automatically. The instance element in the manifest specifies that the service instance should be created in the disabled state.)

The administrator can then use the following commands to enable the `foo` service and to verify that the service is online and that the daemon is running:

```
# svcadm enable foo
# svcs foo
STATE    STIME    FMRI
online   10:57:11 svc:/system/foo:default_foo
# ps -ef | grep foo
    root   753    1  89 10:57:11 ?  0:48 /opt/SUNWsmftest/bin/foo
```

At this point, the `foo` service is ready to handle service requests.

Service Dependencies

Many services require the availability of other services before they are able to start. There are also cases when a service must be restarted if another service is restarted, or stopped when another service has been disabled.

Existing dependency relationships for a service can be viewed using options to the `svcs` command. The "-d" option shows what other services this service depends on, and the "-D" option shows what other services depend on this service. For example, the `sendmail` service has the following dependencies:

```
# svcs -d network/smtp:sendmail
STATE          STIME    FMRI
online         18:20:14 svc:/system/identity:domain
online         18:20:26 svc:/network/service:default
online         18:20:27 svc:/system/filesystem/local:default
online         18:20:27 svc:/milestone/name-services:default
online         18:20:27 svc:/system/system-log:default
online         18:20:30 svc:/system/filesystem/autofs:default
# svcs -D network/smtp:sendmail
STATE          STIME    FMRI
online         18:20:32 svc:/milestone/multi-user:default
```

Entries in a service's XML manifest file are used to describe the dependency relationship between services. An excerpt from the manifest for the `cron` service defines two dependency relationships for `cron` (the `cron` service depends on local filesystem services and name services). A third dependency is defined using the "`dependent`" block, which is used to indicate that the multi-user milestone service depends on the `cron` service.

```
...
<service
    name='system/cron'
    type='service'
    version='1'>

<single_instance />

<dependency
    name='usr'
    type='service'
    grouping='require_all'
    restart_on='none'>
    service_fmri value='svc:/system/filesystem/local' />
</dependency>

<dependency
    name='ns'
    type='service'
    grouping='require_all'
    restart_on='none'>
    service_fmri value='svc:/milestone/name-services' />
</dependency>

<dependent
    name='cron_multi-user'
    type='service'
    grouping='optional_all'
    restart_on='none'>
    service_fmri value='svc:/milestone/multi-user' />
</dependent>
...
```

Dependency Attributes

In an XML manifest like the one above, the "`grouping`" attribute describes the meaning of a dependency with multiple targets. Some of the options are:

- `require_all` means that *all* dependency services must be online before the dependent service will start.
- `require_any` means that *only one* in the group of dependency services must be online for the dependent service to start.
- `optional_all` means that if the dependency services are enabled and able to run (not in maintenance), they must be online or degraded before the dependent service is started.
- `exclude_all` means that all of the dependency services must *not* be running for the dependent service to run.

Another attribute in the XML dependency block is the `restart_on` field, which describes what action the dependent service's restarter will take when one of the services on which it depends changes state. Here is a list of options for the `restart_on` attribute:

- If `restart_on` is set to `none`, the dependency service(s) need only be online for the dependent service to start. After that, the dependency service(s) can be stopped and the dependent service will continue to run.
- If `restart_on` is set to `error` and the dependency service(s) are restarted because the service encountered a hardware error or software error (such as a coredump), then the dependent service is also restarted.
- If `restart_on` is set to `restart` and the dependency service(s) get restarted, then the dependent service will be restarted also. (Note that the process associated with the service instance will be restarted with a different process ID.)
- If `restart_on` is set to `refresh` and the dependency service(s) are restarted or refreshed (via `svcadm refresh svc`) then the dependent service also gets restarted. (Note that the process associated with the service instance will be restarted with a different process ID.)

Service Dependency Example

In the earlier manifest for the `foo` service (page 17), a dependency property (`foo_cron`) is defined and specifies that the `foo` service depends on `system/cron`. The dependency also specifies "`restart_on=refresh`". This means that if the `cron` service gets refreshed or restarted, then `foo` also gets restarted. Table 3 shows how actions on `cron` might impact the `foo` service with `restart_on=refresh`.

Table 3. An example of dependent services with `restart_on=refresh`

Action	Result
`pkill -9 cron`	`cron` gets restarted (this is the Solaris OS default behavior) `foo` also gets restarted
`svcadm disable system/cron`	`cron` is disabled `foo` goes offline
`svcadm enable system/cron`	`cron` goes online `foo` goes online
`svcadm refresh system/cron`	`cron` remains running and its configuration is updated (`cron` has no refresh method) `foo` gets restarted
`pkill -9 foo`	`cron` is not affected `foo` gets restarted

SMF Initialization at Boot

Figure 3 illustrates how SMF is initialized at boot time. As with previous Solaris OS versions, the `init` process is still the first process that the Solaris 10 OS kernel starts at boot. The `init` process reads `/etc/inittab`, which includes an entry for the SMF master restarter daemon `svc.startd`. `svc.startd` starts (and restarts) all SMF services, except for services which specify a delegated restarter (such as `inetd`, which manages network services like `ftp`, `rlogin`, etc.).

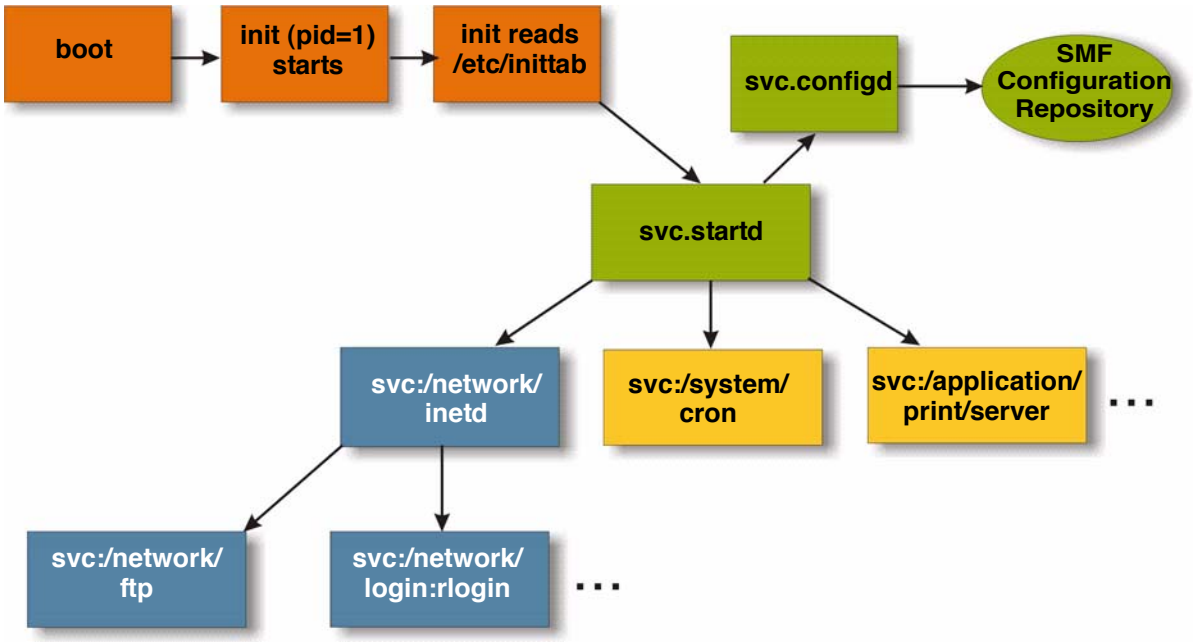

Figure 3. Initialization of SMF services

The SMF master restarter daemon invokes a second daemon, `svc.configd`. This daemon is the SMF configuration daemon, and manages all access to the underlying SMF repository. The repository database resides in the file `/etc/svc/repository.db`. It is backed up automatically and can be recovered if it becomes corrupted (see the description of repository back-up and recovery in Chapter 14, “Managing Services” in the Solaris 10 System Administration Guide: Basic Administration (817-1985-xx).

The repository maintains configuration information for each service, including its restarter and its methods. The restarter specified for each service invokes the appropriate method to start each service.

As services start, there are two directories used for log files. The first directory, `/etc/svc/volatile`, is used by `svc.startd` and records repository errors as well as errors for services that start before `svc:/system/filesystem/minimal` (i.e., before `/var` is mounted). Other service messages are logged to a second log file directory, `/var/svc/log`. For some system services, this is where the restarter directs messages in related log files.

Milestone Services and System Booting

A milestone is a special type of SMF service that merges several service dependencies. Usually, a milestone service does nothing useful itself, but declares a specific state of system-readiness on which other services can depend. The run level-to-milestone relationship is a one-way master-slave model — the run level is the master which, when changed, also changes the milestone. Changing the milestone, however, does not inherently change the run level. To avoid confusion, it is better to continue to use the previously available run level commands (e.g., “`init s`”, “`boot -s`", etc.).

The services that constitute run levels S, 2, and 3 are each represented by milestone services — the single-user, multi-user, and multi-user-server milestones correspond to run levels S, 2, and 3, respectively (Table 4). In addition to the run level milestones, there are two additional milestones: “all” and “none”. These are shorthand for a milestone with no services, or a milestone with all enabled services, respectively.

Table 4. Corresponding milestones and run levels

SMF Milestone	Run Level
milestone none	N/A
milestone single-user	S
milestone multi-user	2
milestone multi-user-server	3
milestone all	3

New boot options allow the system to be booted directly to one of these five specific milestones:

```
boot -m milestone=none
```

Note – The milestone boot options may be useful for system maintenance (particularly when the system will not boot otherwise) and sometimes for testing the addition of new services. Although SMF adds "`boot -m`" and “`svcadm milestone`”, the run level interfaces (e.g., “`init s`”, “`boot -s`", etc.) still exist, and should be used for most system administrative tasks.

In the rare case when a milestone command is used, remember that there are distinct differences between milestones and run levels. For example, booting to the single-user milestone (with “`boot -m milestone=single-user`") is different than the commonly used "`boot -s`". When the system is explicitly booted to the single-user milestone, exiting the console administrative shell with Control-D does not complete the full system boot process (as Control-D does with "`boot -s`"). After the command "`boot -m milestone=single-user`", the command:

```
# svcadm milestone all
```

brings the system to the multi-user-server milestone. Changing the milestone in this way, however, will not change the run level as reported by "`who -r`".

Specifying a Milestone Dependency on a Service

Suppose a new service is to start before a specific milestone is reached — for example, suppose the `foo` service should start before the system reaches `svc:/milestone/single-user`. Using the dependent tag in the manifest file `foo.xml` indicates that the the single-user milestone is dependent on the new service `foo`:

```
...
<!-- single-user milestone is dependent on foo -->
<dependent
      name='foo_single-user'
      grouping='optional_all'
      restart_on='none'>
      <service_fmri value='svc:/milestone/single-user' />
      </dependent>
...
```

The dependent relationship is identified through the name `foo_single-user`. (Note the naming convention indicates that the single-user milestone depends on foo, and helps to avoid name conflicts.) The grouping tag `optional_all` means that if the `foo` service is enabled and able to run (it is not in the maintenance state), then it must be online or degraded before the single-user milestone is considered complete.

Once the manifest file is read, the dependent tag creates a property group in `svc:/milestone/single-user` as reported by `svccfg`:

```
foo_single-user                         dependency
foo_single-user/entities                fmri  svc:/system/foo
foo_single-user/external                boolean  true
foo_single-user/grouping                astring  optional_all
foo_single-user/restart_on              astring  none
foo_single-user/type                    astring  service
```

It also creates a property group in `svc:/system/foo`:

```
dependents                    framework
dependents/foo_single-user     astring  svc:/milestone/single-user
```

By making a milestone service dependent on a new service, then the new service will be started automatically when the run level is changed.

Resolving Dependency Cycles

SMF service dependencies offer a tremendous benefit in making sure that a service is not started unless its requirements are met. However, service dependencies also introduce the potential for system administrators to inadvertently create a new service-related problem — a dependency cycle. A dependency cycle occurs when a service depends on another service which in turn depends on the first service.

For example, suppose an administrator adds a new service (`foo`) by constructing a new service manifest file. The new service is to run before the system reaches single-user mode, so the administrator adds a dependency relationship such that `milestone/single-user` is dependent on `foo` (Figure 4). Since the `foo` service also requires name services running before it can start, the administrator adds a second dependency relationship such that `foo` depends on `name-services`. In this way, the administrator has accidentally created a dependency cycle — `milestone/single-user` cannot complete until the `foo` service starts, but the `foo` service cannot start until `name-services` starts, and `name-services` cannot start because it is waiting on `milestone/single-user` to complete. Figure 4 shows a hypothetical chain of dependencies that might exist in such a situation.

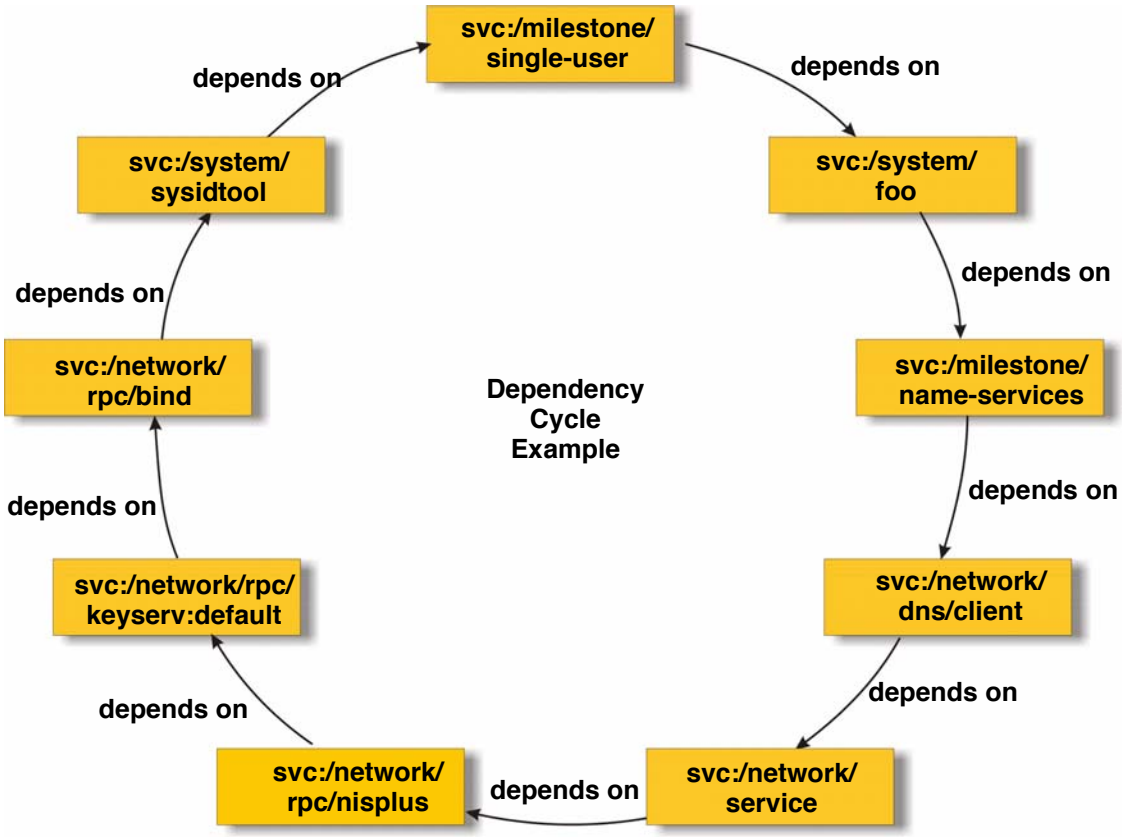

Figure 4. Dependency cycle example

If the system detects a dependency cycle during the import of the manifest file, it may generate an error message as in the following:

```
# svccfg import /var/svc/manifest/system/foo_daemon.xml
# Jan  1 08:08:22 v4u-5k svc.startd[7]: Putting service
svc:/milestone/single-user:default into maintenance because it
completes a dependency cycle:
Jan  1 08:08:22 v4u-5k svc:/system/foo_daemon
Jan  1 08:08:22 v4u-5k svc:/system/foo_daemon:default_foo_daemon
Jan  1 08:08:22 v4u-5k svc:/milestone/name-services
Jan  1 08:08:22 v4u-5k svc:/milestone/name-services:default
Jan  1 08:08:22 v4u-5k svc:/network/dns/client
Jan  1 08:08:22 v4u-5k svc:/network/dns/client:default
Jan  1 08:08:22 v4u-5k svc:/network/service
Jan  1 08:08:22 v4u-5k svc:/network/service:default
Jan  1 08:08:22 v4u-5k svc:/network/rpc/nisplus
Jan  1 08:08:22 v4u-5k svc:/network/rpc/nisplus:default
Jan  1 08:08:22 v4u-5k svc:/network/rpc/keyserv
Jan  1 08:08:22 v4u-5k svc:/network/rpc/keyserv:default
Jan  1 08:08:22 v4u-5k svc:/network/rpc/bind
Jan  1 08:08:22 v4u-5k svc:/network/rpc/bind:default
Jan  1 08:08:22 v4u-5k svc:/system/sysidtool:net
Jan  1 08:08:22 v4u-5k svc:/milestone/single-user:default
```

If a dependency cycle is accidentally created, it is possible that the system will not go into multi-user mode, or it might hang during the boot process. If this occurs, then booting to the “none” milestone may be useful:

```
# boot -m milestone=none
```

After entering the root password, the root user is in a maintenance shell with a read-only root file system. The next command mounts the `/usr` and root partitions in read-write mode so that the problem can be resolved.

```
# svcadm enable -rts filesystem/usr
```

At this stage, the administrator can remove the XML manifest file for the `foo` service, disable the `foo` service, delete its configuration from the repository, and reboot:

```
# rm /var/svc/manifest/system/foo.xml
# svcadm disable foo
# svccfg delete system/foo
# reboot
```

Another Dependency Example — Direct Root Login

The Solaris 10 OS features Role-Based Access Control (RBAC), which allows administrative tasks to be divided among a number of roles, granting each role only the necessary authority needed to perform related administrative tasks. RBAC allows all administrative actions to be traceable to an authenticated individual instead of to just a single root user, which provides more granular user control and greater accountability.

By leveraging RBAC capabilities, an administrator can make a server more secure by specifying root privileges as a role and preventing direct login to the server as the user "root". An administrator might also disable all local logins on the server by modifying `/etc/passwd`. In such a configuration, to gain root access, the administrator must have a valid network login on the corporate network and then assume the privileged role of root. In this way root actions can be subsequently traced back to the actual user.

Although such a configuration improves accountability for root access, a problem can occur if name services such as NIS do not initialize properly in this type of configuration — in such a case, no one, not even root, can log in. (Such an error might occur, for example, if `/etc/defaultdomain` was incorrect or missing.) The system continues to give the usual console login prompt, but every login attempt is answered with a "`Login incorrect`" message.

Wouldn't it be great if the system was smart enough to allow root to login for a maintenance shell when such a problem occurs? That's exactly what this dependency example does. The `system/console-login` service, which displays the console login screen, must be prevented from starting if `network/nis/client` won't start name services. By establishing a dependency relationship (`system/console-login` depends on `network/nis/client`, as shown below), SMF mandates that the console login screen will not start without name services running:

```
# svccfg
svc:> select console-login
svc:/system/console-login> addpg neednis dependency
svc:/system/console-login> setprop neednis/entities = fmri: svc:/network/nis/client
svc:/system/console-login> setprop neednis/grouping = astring: require_all
svc:/system/console-login> setprop neednis/restart_on = astring: none
svc:/system/console-login> setprop neednis/type = astring: service
svc:/system/console-login> quit          (exit the svccfg interactive session)
# svcadm refresh console-login
```

If `svc.startd` cannot start `system/console-login` (in this case, due to `nis/client` failing), then it calls `/etc/sulogin` to enable a maintenance shell (similar to what the administrator sees with "`boot -s`"). If this happens, then the administrator can still log in as root and make repairs.

Summary

The new Service Management Facility (SMF) in the Solaris 10 OS helps to simplify the management of system and application services. The SMF repository maintains service properties even across system reboots, allowing services to be easily and automatically restarted when necessary. In addition, SMF allows administrators to define relationships between services so that a service is not started unless the other services that it needs are already running.

SMF provides consistent management interfaces that create and control both system and application services. It greatly improves visibility into service states and provides error logs that can help administrators more quickly resolve service-related problems. As a part of the innovative Predictive Self-Healing technologies that Sun has engineered into its software and hardware products, SMF capabilities improve overall service management, simplifying data center administration and contributing to lower administrative cost and enhanced availability.

About the Author

Rob Romack is a member of Sun's PTS Americas Midrange Server Group where he currently focuses on resolving issues with Sun's midrange servers. Prior to his current role with PTS engineering, Rob worked for Sun Enterprise Services in various roles including as a System Support Engineer in San Francisco, and in the OS kernel support group in the call center.

References

- Man pages for the Solaris 10 OS, including:
  - `smf(5)`
  - `smf_bootstrap(5)`
  - `smf_method(5)`
  - `svc.startd(1M)`
  - `svcadm(1M)`
  - `svccfg(1M)`
  - `svcprop(1)`
  - `svcs(1)`
  - `service_bundle(4)`
  - `inetadm(1M)`
- Solaris 10 System Administration Guide: Basic Administration (817-1985-xx), Chapter 14, “Managing Services"
- The OpenSolaris SMF community, http://opensolaris.org/os/community/smf/
- “Predictive Self-Healing: Solaris Service Management Facility — Service Developer Introduction,” BigAdmin System Administration Portal, http://www.sun.com/bigadmin/content/selfheal/sdev_intro.html
- “Predictive Self-Healing: Solaris Service Management Facility — Quickstart Guide,” BigAdmin System Administration Portal, http://www.sun.com/bigadmin/content/selfheal/smf-quickstart.html
- “In a Class By Itself — The Solaris™ 10 Operating System”, A Sun Whitepaper, November, 2004.
- Brunette, Glenn. “Restricting Service Administration in the Solaris 10 Operating System,” *Sun BluePrints OnLine*, June 2005. To access this article online, go to http://www.sun.com/blueprints/0605/819-2887.pdf

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals through this program.

Accessing Sun Documentation Online

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com`

To reference Sun BluePrints OnLine articles, visit the Sun BluePrints OnLine Web site at:
`http://www.sun.com/blueprints/online.html`