Chapter

9 Unix Services

Contents

This chapter covers a number of basic services that are common to many Unix systems. All administrators should be familiar with them.

9.1. System Boot

When you boot the computer, the many messages scrolling by on the console display many automatic initializations and configurations that are being executed. Sometimes you may wish to slightly alter how this stage works, which means that you need to understand it well. That is the purpose of this section.

First, the BIOS takes control of the computer, detects the disks, loads the *Master Boot Record*, and executes the bootloader. The bootloader takes over, finds the kernel on the disk, loads and executes it. The kernel is then initialized, and starts to search for and mount the partition containing the root filesystem, and finally executes the first program — `init`. Frequently, this "root partition" and this `init` are, in fact, located in a virtual filesystem that only exists in RAM (hence its name, "initramfs", formerly called "initrd" for "initialization RAM disk"). This filesystem is loaded in memory by the bootloader, often from a file on a hard drive or from the network. It contains the bare minimum required by the kernel to load the "true" root filesystem: this may be driver modules for the hard drive, or other devices without which the system cannot boot, or, more frequently, initialization scripts and modules for assembling RAID arrays, opening encrypted partitions, activating LVM volumes, etc. Once the root partition is mounted, the initramfs hands over control to the real init, and the machine goes back to the standard boot process.

9.1.1. The systemd init system

The "real init" is currently provided by *systemd* and this section documents this init system.

CULTURE
Before systemd

`systemd` is a relatively recent "init system", and although it was already available, to a certain extent, in *Wheezy*, it has only become the default in Debian *Jessie*. Previous releases relied, by default, on the "System V init" (in the *sysv-rc* package), a much more traditional system. We describe the System V init later on.

ALTERNATIVE
Other boot systems

This book describes the boot system used by default in Debian *Jessie* (as implemented by the *systemd* package), as well as the previous default, *sysvinit*, which is derived and inherited from *System V* Unix systems; there are others.

file-rc is a boot system with a very simple process. It keeps the principle of runlevels, but replaces the directories and symbolic links with a configuration file, which indicates to `init` the processes that must be started and their launch order.

The `upstart` system is still not perfectly tested on Debian. It is event based: init scripts are no longer executed in a sequential order but in response to events such as the completion of another script upon which they are dependent. This system, started by Ubuntu, is present in Debian *Jessie*, but is not the default; it comes, in fact, as a replacement for *sysvinit*, and one of the tasks launched by `upstart` is to launch the scripts written for traditional systems, especially those from the *sysv-rc* package.

There are also other systems and other operating modes, such as `runit` or `minit`, but they are relatively specialized and not widespread.

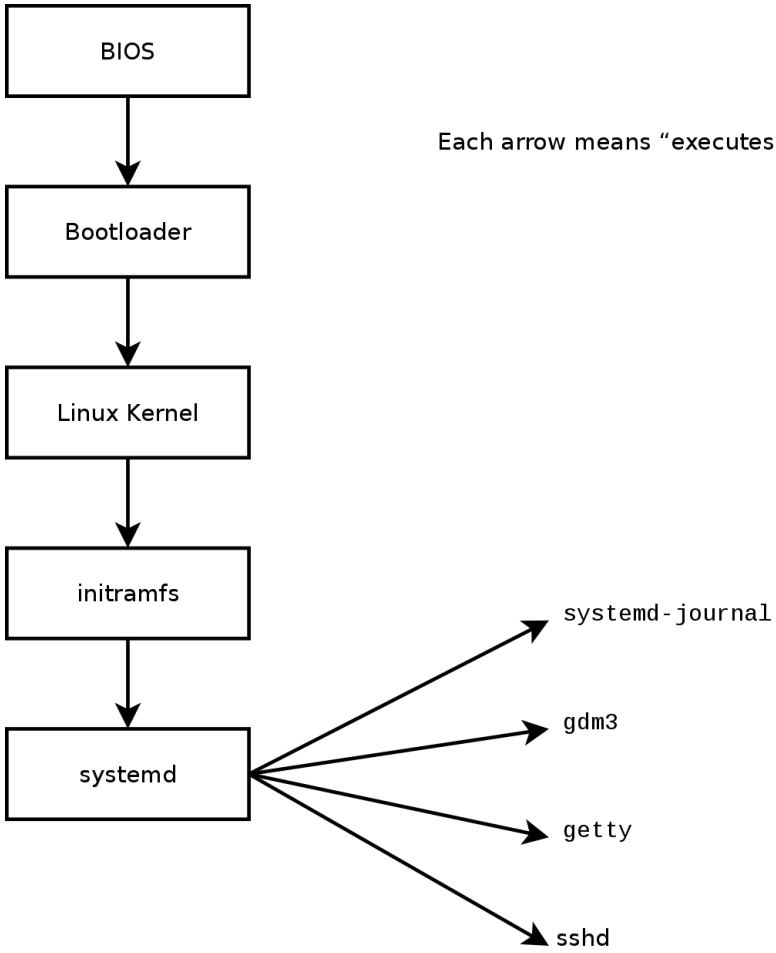

Figure 9.1 *Boot sequence of a computer running Linux with systemd*

SPECIFIC CASE

Booting from the network

In some configurations, the BIOS may be configured not to execute the MBR, but to seek its equivalent on the network, making it possible to build computers without a hard drive, or which are completely reinstalled on each boot. This option is not available on all hardware and it generally requires an appropriate combination of BIOS and network card.

Booting from the network can be used to launch the `debian-installer` or FAI (see section 4.1, “Installation Methods” page 48).

BACK TO BASICS

The process, a program instance

A process is the representation in memory of a running program. It includes all of the information necessary for the proper execution of the software (the code itself, but also the data that it has in memory, the list of files that it has opened, the network connections it has established, etc.). A single program may be instantiated into several processes, not necessarily running under different user IDs.

SECURITY

Using a shell as `init` to gain root rights

By convention, the first process that is booted is the `init` program (which is a symbolic link to `/lib/systemd/systemd` by default). However, it is possible to pass an `init` option to the kernel indicating a different program.

Any person who is able to access the computer can press the Reset button, and thus reboot it. Then, at the bootloader's prompt, it is possible to pass the `init=/bin/sh` option to the kernel to gain root access without knowing the administrator's password.

To prevent this, you can protect the bootloader itself with a password. You might also think about protecting access to the BIOS (a password protection mechanism is almost always available), without which a malicious intruder could still boot the machine on a removable media containing its own Linux system, which they could then use to access data on the computer's hard drives.

Finally, be aware that most BIOS have a generic password available. Initially intended for troubleshooting for those who have forgotten their password, these passwords are now public and available on the Internet (see for yourself by searching for "generic BIOS passwords" in a search engine). All of these protections will thus impede unauthorized access to the machine without being able to completely prevent it. There is no reliable way to protect a computer if the attacker can physically access it; they could dismount the hard drives to connect them to a computer under their own control anyway, or even steal the entire machine, or erase the BIOS memory to reset the password...

Systemd executes several processes, in charge of setting up the system: keyboard, drivers, filesystems, network, services. It does this while keeping a global view of the system as a whole, and the requirements of the components. Each component is described by a "unit file" (sometimes more); the general syntax is derived from the widely-used "*.ini files" syntax, with *key* =*value* pairs grouped between [*section*] headers. Unit files are stored under `/lib/systemd/system/` and `/etc/systemd/system/`; they come in several flavours, but we will focus on "services" and "targets" here.

A systemd "service file" describes a process managed by systemd. It contains roughly the same information as old-style init-scripts, but expressed in a declaratory (and much more concise) way. Systemd handles the bulk of the repetitive tasks (starting and stopping the process, checking its status, logging, dropping privileges, and so on), and the service file only needs to fill in the specifics of the process. For instance, here is the service file for SSH:

```
[Unit]
Description=OpenBSD Secure Shell server
After=network.target auditd.service
ConditionPathExists=!/etc/ssh/sshd_not_to_be_run

[Service]
EnvironmentFile=-/etc/default/ssh
ExecStart=/usr/sbin/sshd -D $SSHD_OPTS
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=on-failure
```

```
[Install]
WantedBy=multi-user.target
Alias=sshd.service
```

As you can see, there is very little code in there, only declarations. Systemd takes care of displaying progress reports, keeping track of the processes, and even restarting them when needed.

A systemd "target file" describes a state of the system, where a set of services are known to be operational. It can be thought of as an equivalent of the old-style runlevel. One of the targets is local-fs.target; when it is reached, the rest of the system can assume that all local filesystems are mounted and accessible. Other targets include network-online.target and sound.target. The dependencies of a target can be listed either within the target file (in the Requires= line), or using a symbolic link to a service file in the /lib/systemd/system/*targetname*.target.wants/ directory. For instance, `/etc/systemd/system/printer.target.wants/` contains a link to `/lib/systemd/system/cups.service`; systemd will therefore ensure CUPS is running in order to reach printer.target.

Since unit files are declarative rather than scripts or programs, they cannot be run directly, and they are only interpreted by systemd; several utilities therefore allow the administrator to interact with systemd and control the state of the system and of each component.

The first such utility is `systemctl`. When run without any arguments, it lists all the unit files known to systemd (except those that have been disabled), as well as their status. `systemctl status` gives a better view of the services, as well as the related processes. If given the name of a service (as in `systemctl status ntp.service`), it returns even more details, as well as the last few log lines related to the service (more on that later).

Starting a service by hand is a simple matter of running `systemctl start` *`servicename`*`.service`. As one can guess, stopping the service is done with `systemctl stop` *`servicename`*`.service`; other subcommands include `reload` and `restart`.

To control whether a service is active (i.e. whether it will get started automatically on boot), use `systemctl enable` *`servicename`*`.service` (or `disable`). `is-enabled` allows checking the status of the service.

An interesting feature of systemd is that it includes a logging component named `journald`. It comes as a complement to more traditional logging systems such as `syslogd`, but it adds interesting features such as a formal link between a service and the messages it generates, and the ability to capture error messages generated by its initialisation sequence. The messages can be displayed later on, with a little help from the `journalctl` command. Without any arguments, it simply spews all log messages that occurred since system boot; it will rarely be used in such a manner. Most of the time, it will be used with a service identifier:

```
# journalctl -u ssh.service
-- Logs begin at Tue 2015-03-31 10:08:49 CEST, end at Tue 2015-03-31 17:06:02 CEST.
       --
Mar 31 10:08:55 mirtuel sshd[430]: Server listening on 0.0.0.0 port 22.
Mar 31 10:08:55 mirtuel sshd[430]: Server listening on :: port 22.
Mar 31 10:09:00 mirtuel sshd[430]: Received SIGHUP; restarting.
```

```
Mar 31 10:09:00 mirtuel sshd[430]: Server listening on 0.0.0.0 port 22.
Mar 31 10:09:00 mirtuel sshd[430]: Server listening on :: port 22.
Mar 31 10:09:32 mirtuel sshd[1151]: Accepted password for roland from 192.168.1.129
       port 53394 ssh2
Mar 31 10:09:32 mirtuel sshd[1151]: pam_unix(sshd:session): session opened for user
       roland by (uid=0)
```

Another useful command-line flag is `-f`, which instructs `journalctl` to keep displaying new messages as they are emitted (much in the manner of `tail -f` *`file`*).

If a service doesn't seem to be working as expected, the first step to solve the problem is to check that the service is actually running with `systemctl status`; if it is not, and the messages given by the first command are not enough to diagnose the problem, check the logs gathered by journald about that service. For instance, assume the SSH server doesn't work:

```
# systemctl status ssh.service
● ssh.service - OpenBSD Secure Shell server
   Loaded: loaded (/lib/systemd/system/ssh.service; enabled)
   Active: failed (Result: start-limit) since Tue 2015-03-31 17:30:36 CEST; 1s ago
  Process: 1023 ExecReload=/bin/kill -HUP $MAINPID (code=exited, status=0/SUCCESS)
  Process: 1188 ExecStart=/usr/sbin/sshd -D $SSHD_OPTS (code=exited, status=255)
 Main PID: 1188 (code=exited, status=255)

Mar 31 17:30:36 mirtuel systemd[1]: ssh.service: main process exited, code=exited,
       status=255/n/a
Mar 31 17:30:36 mirtuel systemd[1]: Unit ssh.service entered failed state.
Mar 31 17:30:36 mirtuel systemd[1]: ssh.service start request repeated too quickly,
       refusing to start.
Mar 31 17:30:36 mirtuel systemd[1]: Failed to start OpenBSD Secure Shell server.
Mar 31 17:30:36 mirtuel systemd[1]: Unit ssh.service entered failed state.
# journalctl -u ssh.service
-- Logs begin at Tue 2015-03-31 17:29:27 CEST, end at Tue 2015-03-31 17:30:36 CEST.
       --
Mar 31 17:29:27 mirtuel sshd[424]: Server listening on 0.0.0.0 port 22.
Mar 31 17:29:27 mirtuel sshd[424]: Server listening on :: port 22.
Mar 31 17:29:29 mirtuel sshd[424]: Received SIGHUP; restarting.
Mar 31 17:29:29 mirtuel sshd[424]: Server listening on 0.0.0.0 port 22.
Mar 31 17:29:29 mirtuel sshd[424]: Server listening on :: port 22.
Mar 31 17:30:10 mirtuel sshd[1147]: Accepted password for roland from 192.168.1.129
       port 38742 ssh2
Mar 31 17:30:10 mirtuel sshd[1147]: pam_unix(sshd:session): session opened for user
       roland by (uid=0)
Mar 31 17:30:35 mirtuel sshd[1180]: /etc/ssh/sshd_config line 28: unsupported option
       "yess".
Mar 31 17:30:35 mirtuel systemd[1]: ssh.service: main process exited, code=exited,
       status=255/n/a
Mar 31 17:30:35 mirtuel systemd[1]: Unit ssh.service entered failed state.
Mar 31 17:30:35 mirtuel sshd[1182]: /etc/ssh/sshd_config line 28: unsupported option
       "yess".
```

```
Mar 31 17:30:35 mirtuel systemd[1]: ssh.service: main process exited, code=exited,
       status=255/n/a
Mar 31 17:30:35 mirtuel systemd[1]: Unit ssh.service entered failed state.
Mar 31 17:30:35 mirtuel sshd[1184]: /etc/ssh/sshd_config line 28: unsupported option
       "yess".
Mar 31 17:30:35 mirtuel systemd[1]: ssh.service: main process exited, code=exited,
       status=255/n/a
Mar 31 17:30:35 mirtuel systemd[1]: Unit ssh.service entered failed state.
Mar 31 17:30:36 mirtuel sshd[1186]: /etc/ssh/sshd_config line 28: unsupported option
       "yess".
Mar 31 17:30:36 mirtuel systemd[1]: ssh.service: main process exited, code=exited,
       status=255/n/a
Mar 31 17:30:36 mirtuel systemd[1]: Unit ssh.service entered failed state.
Mar 31 17:30:36 mirtuel sshd[1188]: /etc/ssh/sshd_config line 28: unsupported option
       "yess".
Mar 31 17:30:36 mirtuel systemd[1]: ssh.service: main process exited, code=exited,
       status=255/n/a
Mar 31 17:30:36 mirtuel systemd[1]: Unit ssh.service entered failed state.
Mar 31 17:30:36 mirtuel systemd[1]: ssh.service start request repeated too quickly,
       refusing to start.
Mar 31 17:30:36 mirtuel systemd[1]: Failed to start OpenBSD Secure Shell server.
Mar 31 17:30:36 mirtuel systemd[1]: Unit ssh.service entered failed state.
# vi /etc/ssh/sshd_config
# systemctl start ssh.service
# systemctl status ssh.service
● ssh.service - OpenBSD Secure Shell server
   Loaded: loaded (/lib/systemd/system/ssh.service; enabled)
   Active: active (running) since Tue 2015-03-31 17:31:09 CEST; 2s ago
  Process: 1023 ExecReload=/bin/kill -HUP $MAINPID (code=exited, status=0/SUCCESS)
 Main PID: 1222 (sshd)
   CGroup: /system.slice/ssh.service
           └─1222 /usr/sbin/sshd -D
#
```

After checking the status of the service (failed), we went on to check the logs; they indicate an error in the configuration file. After editing the configuration file and fixing the error, we restart the service, then verify that it is indeed running.

GOING FURTHER

Other types of unit files

We have only described the most basic of systemd's capabilities in this section. It offers many other interesting features; we will only list a few here:

- socket activation: a "socket" unit file can be used to describe a network or Unix socket managed by systemd; this means that the socket will be created by systemd, and the actual service may be started on demand when an actual connection attempt comes. This roughly replicates the feature set of `inetd`. See `systemd.socket(5)`.
- timers: a "timer" unit file describes events that occur with a fixed frequency or on specific times; when a service is linked to such a timer, the corresponding task will be executed whenever the timer fires. This allows replicating part of the `cron` features. See `systemd.timer(5)`.

- network: a "network" unit file describes a network interface, which allows configuring such interfaces as well as expressing that a service depends on one particular interface being up.

9.1.2. The System V init system

The System V init system (which we'll call init for brevity) executes several processes, following instructions from the `/etc/inittab` file. The first program that is executed (which corresponds to the *sysinit* step) is `/etc/init.d/rcS`, a script that executes all of the programs in the `/etc/rcS.d/` directory.

Among these, you will find successively programs in charge of:

- configuring the console's keyboard;
- loading drivers: most of the kernel modules are loaded by the kernel itself as the hardware is detected; extra drivers are then loaded automatically when the corresponding modules are listed in `/etc/modules`;
- checking the integrity of filesystems;
- mounting local partitions;
- configuring the network;
- mounting network filesystems (NFS).

BACK TO BASICS

Kernel modules and options

Kernel modules also have options that can be configured by putting some files in `/etc/modprobe.d/`. These options are defined with directives like this: `options` *`module-name option-name=option-value`*. Several options can be specified with a single directive if necessary.

These configuration files are intended for `modprobe` — the program that loads a kernel module with its dependencies (modules can indeed call other modules). This program is provided by the *kmod* package.

After this stage, `init` takes over and starts the programs enabled in the default runlevel (which is usually runlevel 2). It executes `/etc/init.d/rc 2`, a script that starts all services which are listed in `/etc/rc2.d/` and whose names start with the "S" letter. The two-figures number that follows had historically been used to define the order in which services had to be started, but nowadays the default boot system uses `insserv`, which schedules everything automatically based on the scripts' dependencies. Each boot script thus declares the conditions that must be met to start or stop the service (for example, if it must start before or after another service); `init` then launches them in the order that meets these conditions. The static numbering of scripts is therefore no longer taken into consideration (but they must always have a name beginning with "S" followed by two digits and the actual name of the script used for the dependencies). Generally, base services (such as logging with `rsyslog`, or port assignment with `portmap`) are started first, followed by standard services and the graphical interface (`gdm3`).

This dependency-based boot system makes it possible to automate re-numbering, which could be rather tedious if it had to be done manually, and it limits the risks of human error, since scheduling is conducted according to the parameters that are indicated. Another benefit is that services can be started in parallel when they are independent from one another, which can accelerate the boot process.

`init` distinguishes several runlevels, so it can switch from one to another with the `telinit` *`new-level`* command. Immediately, `init` executes `/etc/init.d/rc` again with the new runlevel. This script will then start the missing services and stop those that are no longer desired. To do this, it refers to the content of the `/etc/rc`*X*`.d` (where *X* represents the new runlevel). Scripts starting with "S" (as in "Start") are services to be started; those starting with "K" (as in "Kill") are the services to be stopped. The script does not start any service that was already active in the previous runlevel.

By default, System V init in Debian uses four different runlevels:

- Level 0 is only used temporarily, while the computer is powering down. As such, it only contains many "K" scripts.

- Level 1, also known as single-user mode, corresponds to the system in degraded mode; it includes only basic services, and is intended for maintenance operations where interactions with ordinary users are not desired.

- Level 2 is the level for normal operation, which includes networking services, a graphical interface, user logins, etc.

- Level 6 is similar to level 0, except that it is used during the shutdown phase that precedes a reboot.

Other levels exist, especially 3 to 5. By default they are configured to operate the same way as level 2, but the administrator can modify them (by adding or deleting scripts in the corresponding `/etc/rc`*X*`.d` directories) to adapt them to particular needs.

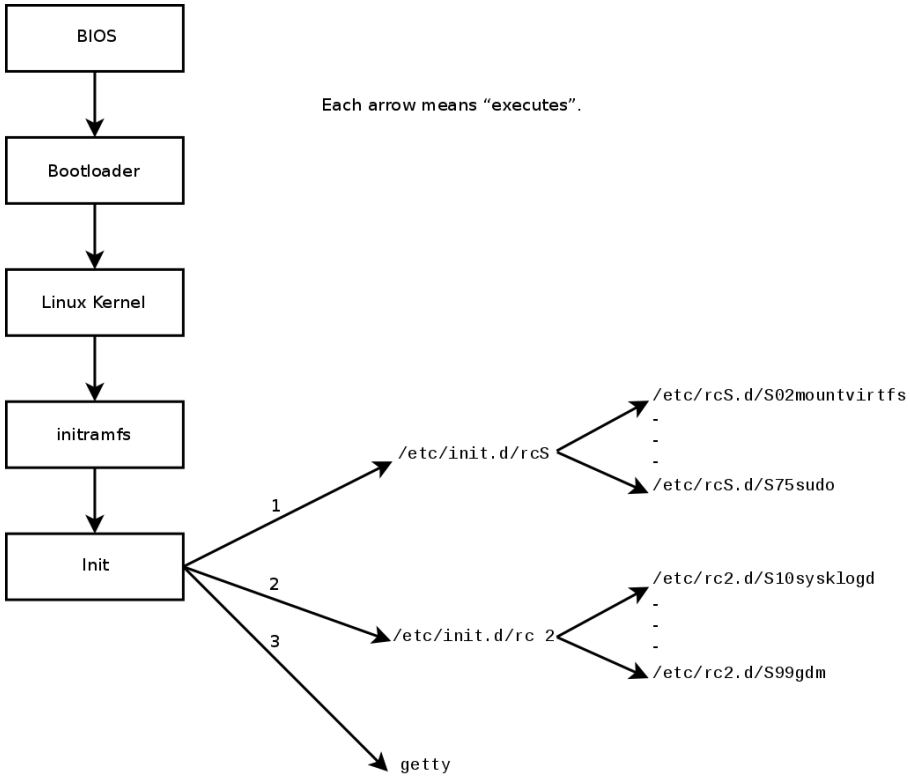

Figure 9.2 *Boot sequence of a computer running Linux with System V init*

All the scripts contained in the various `/etc/rc`*X*`.d` directories are really only symbolic links — created upon package installation by the `update-rc.d` program — pointing to the actual scripts which are stored in `/etc/init.d/`. The administrator can fine tune the services available in each runlevel by re-running `update-rc.d` with adjusted parameters. The `update-rc.d(1)` manual page describes the syntax in detail. Please note that removing all symbolic links (with the remove parameter) is not a good method to disable a service. Instead you should simply configure it to not start in the desired runlevel (while preserving the corresponding calls to stop it in the event that the service runs in the previous runlevel). Since `update-rc.d` has a somewhat convoluted interface, you may prefer using `rcconf` (from the *rcconf* package) which provides a more user-friendly interface.

DEBIAN POLICY

Restarting services

The maintainer scripts for Debian packages will sometimes restart certain services to ensure their availability or get them to take certain options into account. The command that controls a service — `service` *`service operation`* — doesn't take runlevel into consideration, assumes (wrongly) that the service is currently being used, and may thus initiate incorrect operations (starting a service that was deliberately stopped, or stopping a service that is already stopped, etc.). Debian therefore introduced the `invoke-rc.d` program: this program must be used by maintainer scripts to run services initialization scripts and it will only execute the necessary commands. Note that, contrary to common usage, the `.d` suffix is used here in a program name, and not in a directory.

Finally, `init` starts control programs for various virtual consoles (`getty`). It displays a prompt, waiting for a username, then executes `login` *`user`* to initiate a session.

VOCABULARY

Console and terminal

The first computers were usually separated into several, very large parts: the storage enclosure and the central processing unit were separate from the peripheral devices used by the operators to control them. These were part of a separate furniture, the "console". This term was retained, but its meaning has changed. It has become more or less synonymous with "terminal", being a keyboard and a screen.

With the development of computers, operating systems have offered several virtual consoles to allow for several independent sessions at the same time, even if there is only one keyboard and screen. Most GNU/Linux systems offer six virtual consoles (in text mode), accessible by typing the key combinations Control+Alt+F1 through Control+Alt+F6.

By extension, the terms "console" and "terminal" can also refer to a terminal emulator in a graphical X11 session (such as `xterm`, `gnome-terminal` or `konsole`).

9.2. Remote Login

It is essential for an administrator to be able to connect to a computer remotely. Servers, confined in their own room, are rarely equipped with permanent keyboards and monitors — but they are connected to the network.

BACK TO BASICS

Client, server

A system where several processes communicate with each other is often described with the "client/server" metaphor. The server is the program that takes requests coming from a client and executes them. It is the client that controls operations, the server doesn't take any initiative of its own.

9.2.1. Secure Remote Login: SSH

The *SSH* (Secure SHell) protocol was designed with security and reliability in mind. Connections using SSH are secure: the partner is authenticated and all data exchanges are encrypted.

VOCABULARY

Authentication, encryption

When you need to give a client the ability to conduct or trigger actions on a server, security is important. You must ensure the identity of the client; this is authentication. This identity usually consists of a password that must be kept secret, or any other client could get the password. This is the purpose of encryption, which is a form of encoding that allows two systems to communicate confidential information on a public channel while protecting it from being readable to others.

Authentication and encryption are often mentioned together, both because they are frequently used together, and because they are usually implemented with similar mathematical concepts.

CULTURE

Telnet and RSH are obsolete

Before SSH, *Telnet* and *RSH* were the main tools used to login remotely. They are now largely obsolete and should no longer be used even if Debian still provides them.

SSH also offers two file transfer services. `scp` is a command line tool that can be used like `cp`, except that any path to another machine is prefixed with the machine's name, followed by a colon.

```
$ scp file machine:/tmp/
```

`sftp` is an interactive command, similar to `ftp`. In a single session, `sftp` can transfer several files, and it is possible to manipulate remote files with it (delete, rename, change permissions, etc.).

Debian uses OpenSSH, a free version of SSH maintained by the `OpenBSD` project (a free operating system based on the BSD kernel, focused on security) and fork of the original SSH software developed by the SSH Communications Security Corp company, of Finland. This company initially developed SSH as free software, but eventually decided to continue its development under a proprietary license. The OpenBSD project then created OpenSSH to maintain a free version of SSH.

BACK TO BASICS

Fork

A "fork", in the software field, means a new project that starts as a clone of an existing project, and that will compete with it. From there on, both software will usually quickly diverge in terms of new developments. A fork is often the result of disagreements within the development team.

The option to fork a project is a direct result of the very nature of free software; a fork is a healthy event when it enables the continuation of a project as free software (for example in case of license changes). A fork arising from technical or personal disagreements is often a waste of human resources; another resolution would be preferable. Mergers of two projects that previously went through a prior fork are not unheard of.

OpenSSH is split into two packages: the client part is in the *openssh-client* package, and the server is in the *openssh-server* package. The *ssh* meta-package depends on both parts and facilitates installation of both (`apt install ssh`).

Key-Based Authentication

Each time someone logs in over SSH, the remote server asks for a password to authenticate the user. This can be problematic if you want to automate a connection, or if you use a tool that requires frequent connections over SSH. This is why SSH offers a key-based authentication system.

The user generates a key pair on the client machine with `ssh-keygen -t rsa`; the public key is stored in `~/.ssh/id_rsa.pub`, while the corresponding private key is stored in `~/.ssh/id_rsa`.

The user then uses `ssh-copy-id` *`server`* to add their public key to the `~/.ssh/authorized_keys` file on the server. If the private key was not protected with a "passphrase" at the time of its creation, all subsequent logins on the server will work without a password. Otherwise, the private key must be decrypted each time by entering the passphrase. Fortunately, `ssh-agent` allows us to keep private keys in memory to not have to regularly re-enter the password. For this, you simply use `ssh-add` (once per work session) provided that the session is already associated with a functional instance of `ssh-agent`. Debian activates it by default in graphical sessions, but this can be deactivated by changing `/etc/X11/Xsession.options`. For a console session, you can manually start it with `eval $(ssh-agent)`.

CULTURE

OpenSSL flaw in Debian *Etch*

The OpenSSL library, as initially provided in Debian *Etch*, had a serious problem in its random number generator (RNG). Indeed, the Debian maintainer had made a change so that applications using it would no longer generate warnings when analyzed by memory testing tools like `valgrind`. Unfortunately, this change also meant that the RNG was employing only one source of entropy corresponding to the process number (PID) whose 32,000 possible values do not offer enough randomness.

➨ `http://www.debian.org/security/2008/dsa-1571`

Specifically, whenever OpenSSL was used to generate a key, it always produced a key within a known set of hundreds of thousands of keys (32,000 multiplied by a small number of key lengths). This affected SSH keys, SSL keys, and X.509 certificates used by numerous applications, such as OpenVPN. A cracker had only to try all of the keys to gain unauthorized access. To reduce the impact of the problem, the SSH daemon was modified to refuse problematic keys that are listed in the *openssh-blacklist* and *openssh-blacklist-extra* packages. Additionally, the `ssh-vulnkey` command allows identification of possibly compromised keys in the system.

A more thorough analysis of this incident brings to light that it is the result of multiple (small) problems, both within the OpenSSL project and with the Debian package maintainer. A widely used library like OpenSSL should — without modifications — not generate warnings when tested by `valgrind`. Furthermore, the code (especially the parts as sensitive as the RNG) should be better commented to prevent such errors. On Debian's side, the maintainer wanted to validate the modifications with the OpenSSL developers, but simply explained the modifications without providing the corresponding patch to review and failed to mention his role within Debian. Finally, the maintenance choices were sub-optimal: the changes made to the original code were not clearly documented; all the modifications were effectively stored in a Subversion repository, but they ended up all lumped into one single patch during creation of the source package.

It is difficult under such conditions to find the corrective measures to prevent such incidents from recurring. The lesson to be learned here is that every divergence Debian introduces to upstream software must be justified, documented, submitted to the upstream project when possible, and widely publicized. It is from this perspective that the new source package format ("3.0 (quilt)") and the Debian sources webservice were developed.

➨ `http://sources.debian.net`

SECURITY

Protection of the private key

Whoever has the private key can login on the account thus configured. This is why access to the private key is protected by a "passphrase". Someone who acquires a copy of a private key file (for example, `~/.ssh/id_rsa`) still has to know this phrase in order to be able to use it. This additional protection is not, however, impregnable, and if you think that this file has been compromised, it is best to disable that key on the computers in which it has been installed (by removing it from the `authorized_keys` files) and replacing it with a newly generated key.

Using Remote X11 Applications

The SSH protocol allows forwarding of graphical data ("X11" session, from the name of the most widespread graphical system in Unix); the server then keeps a dedicated channel for those data. Specifically, a graphical program executed remotely can be displayed on the X.org server of the local screen, and the whole session (input and display) will be secure. Since this feature allows remote applications to interfere with the local system, it is disabled by default. You can enable it by specifying X11Forwarding yes in the server configuration file (`/etc/ssh/sshd_config`). Finally, the user must also request it by adding the -X option to the `ssh` command-line.

Creating Encrypted Tunnels with Port Forwarding

Its -R and -L options allow `ssh` to create "encrypted tunnels" between two machines, securely forwarding a local TCP port (see sidebar "TCP/UDP" page 222) to a remote machine or vice versa.

VOCABULARY

Tunnel

The Internet, and most LANs that are connected to it, operate in packet mode and not in connected mode, meaning that a packet issued from one computer to another is going to be stopped at several intermediary routers to find its way to its destination. You can still simulate a connected operation where the stream is encapsulated in normal IP packets. These packets follow their usual route, but the stream is reconstructed unchanged at the destination. We call this a "tunnel", analogous to a road tunnel in which vehicles drive directly from the entrance (input) to the exit (output) without encountering any intersections, as opposed to a path on the surface that would involve intersections and changing direction.

You can use this opportunity to add encryption to the tunnel: the stream that flows through it is then unrecognizable from the outside, but it is returned in decrypted form at the exit of the tunnel.

`ssh -L 8000:server:25 intermediary` establishes an SSH session with the *intermediary* host and listens to local port 8000 (see Figure 9.3, "Forwarding a local port with SSH" page 195). For any connection established on this port, `ssh` will initiate a connection from the *intermediary* computer to port 25 on the *server*, and will bind both connections together.

`ssh -R 8000:server:25 intermediary` also establishes an SSH session to the *intermediary* computer, but it is on this machine that `ssh` listens to port 8000 (see Figure 9.4, "Forwarding a remote port with SSH" page 195). Any connection established on this port will cause `ssh` to

open a connection from the local machine on to port 25 of the *server*, and to bind both connections together.

In both cases, connections are made to port 25 on the *server* host, which pass through the SSH tunnel established between the local machine and the *intermediary* machine. In the first case, the entrance to the tunnel is local port 8000, and the data move towards the *intermediary* machine before being directed to the *server* on the "public" network. In the second case, the input and output in the tunnel are reversed; the entrance is port 8000 on the *intermediary* machine, the output is on the local host, and the data are then directed to the *server*. In practice, the server is usually either the local machine or the intermediary. That way SSH secures the connection from one end to the other.

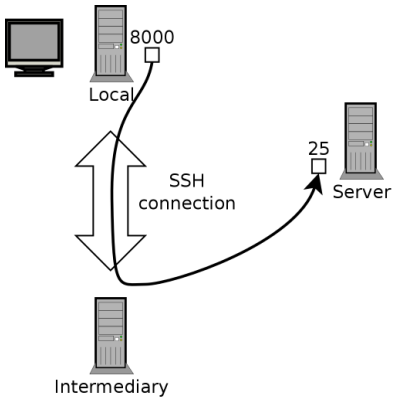

Figure 9.3 *Forwarding a local port with SSH*

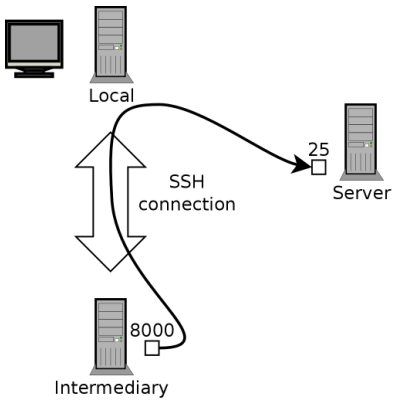

Figure 9.4 *Forwarding a remote port with SSH*

9.2.2. Using Remote Graphical Desktops

VNC (Virtual Network Computing) allows remote access to graphical desktops.

This tool is mostly used for technical assistance; the administrator can see the errors that the user is facing, and show them the correct course of action without having to stand by them.

First, the user must authorize sharing their session. The GNOME graphical desktop environment in *Jessie* includes that option in its configuration panel (contrary to previous versions of Debian, where the user had to install and run `vino`). KDE still requires using `krfb` to allow sharing an existing session over VNC. For other graphical desktop environments, the `x11vnc` command (from the Debian package of the same name) serves the same purpose; you can make it available to the user with an explicit icon.

When the graphical session is made available by VNC, the administrator must connect to it with a VNC client. GNOME has `vinagre` and `remmina` for that, while KDE includes `krdc` (in the menu at K Internet Remote Desktop Client). There are other VNC clients that use the command line, such as `xvnc4viewer` in the Debian package of the same name. Once connected, the administrator can see what is going on, work on the machine remotely, and show the user how to proceed.

SECURITY

VNC over SSH

If you want to connect by VNC, and you don't want your data sent in clear text on the network, it is possible to encapsulate the data in an SSH tunnel (see section 9.2.1.3, "Creating Encrypted Tunnels with Port Forwarding" page 194). You simply have to know that VNC uses port 5900 by default for the first screen (called "localhost:0"), 5901 for the second (called "localhost:1"), etc.

The `ssh -L localhost:5901:localhost:5900 -N -T` *`machine`* command creates a tunnel between local port 5901 in the localhost interface and port 5900 of the *machine* host. The first "localhost" restricts SSH to listening to only that interface on the local machine. The second "localhost" indicates the interface on the remote machine which will receive the network traffic entering in "localhost:5901". Thus `vncviewer localhost:1` will connect the VNC client to the remote screen, even though you indicate the name of the local machine.

When the VNC session is closed, remember to close the tunnel by also quitting the corresponding SSH session.

BACK TO BASICS

Display manager

`gdm3`, `kdm`, `lightdm`, and `xdm` are Display Managers. They take control of the graphical interface shortly after boot in order to provide the user a login screen. Once the user has logged in, they execute the programs needed to start a graphical work session.

VNC also works for mobile users, or company executives, who occasionally need to login from their home to access a remote desktop similar to the one they use at work. The configuration of such a service is more complicated: you first install the *vnc4server* package, change the configuration of the display manager to accept XDMCP Query requests (for `gdm3`, this can be done by adding `Enable=true` in the "xdmcp" section of `/etc/gdm3/daemon.conf`), and finally, start the VNC server with `inetd` so that a session is automatically started when a user tries to login. For example, you may add this line to `/etc/inetd.conf`:

```
5950  stream  tcp  nowait  nobody.tty  /usr/bin/Xvnc Xvnc -inetd -query localhost -
      once -geometry 1024x768 -depth 16 securitytypes=none
```

Redirecting incoming connections to the display manager solves the problem of authentication, because only users with local accounts will pass the `gdm3` login screen (or equivalent `kdm`, `xdm`, etc.). As this operation allows multiple simultaneous logins without any problem (provided the server is powerful enough), it can even be used to provide complete desktops for mobile users (or for less powerful desktop systems, configured as thin clients). Users simply login to the server's screen with `vncviewer` *`server`*`:50`, because the port used is 5950.

9.3. Managing Rights

Linux is definitely a multi-user system, so it is necessary to provide a permission system to control the set of authorized operations on files and directories, which includes all the system resources and devices (on a Unix system, any device is represented by a file or directory). This principle is common to all Unix systems, but a reminder is always useful, especially as there are some interesting and relatively unknown advanced uses.

Each file or directory has specific permissions for three categories of users:

- its owner (symbolized by `u` as in "user");
- its owner group (symbolized by `g` as in "group"), representing all the members of the group;
- the others (symbolized by `o` as in "other").

Three types of rights can be combined:

- reading (symbolized by `r` as in "read");
- writing (or modifying, symbolized by `w` as in "write");
- executing (symbolized by `x` as in "eXecute").

In the case of a file, these rights are easily understood: read access allows reading the content (including copying), write access allows changing it, and execute access allows you to run it (which will only work if it is a program).

SECURITY

`setuid` and `setgid` executables

Two particular rights are relevant to executable files: `setuid` and `setgid` (symbolized with the letter "s"). Note that we frequently speak of "bit", since each of these boolean values can be represented by a 0 or a 1. These two rights allow any user to execute the program with the rights of the owner or the group, respectively. This mechanism grants access to features requiring higher level permissions than those you would usually have.

Since a `setuid` root program is systematically run under the super-user identity, it is very important to ensure it is secure and reliable. Indeed, a user who would manage to subvert it to call a command of their choice could then impersonate the root user and have all rights on the system.

A directory is handled differently. Read access gives the right to consult the list of its entries (files and directories), write access allows creating or deleting files, and execute access allows

crossing through it (especially to go there with the `cd` command). Being able to cross through a directory without being able to read it gives permission to access the entries therein that are known by name, but not to find them if you do not know their existence or their exact name.

SECURITY

`setgid` directory and *sticky bit*

The `setgid` bit also applies to directories. Any newly-created item in such directories is automatically assigned the owner group of the parent directory, instead of inheriting the creator's main group as usual. This setup avoids the user having to change its main group (with the `newgrp` command) when working in a file tree shared between several users of the same dedicated group.

The "sticky" bit (symbolized by the letter "t") is a permission that is only useful in directories. It is especially used for temporary directories where everybody has write access (such as `/tmp/`): it restricts deletion of files so that only their owner (or the owner of the parent directory) can do it. Lacking this, everyone could delete other users' files in `/tmp/`.

Three commands control the permissions associated with a file:

- `chown` *`user file`* changes the owner of the file;
- `chgrp` *`group file`* alters the owner group;
- `chmod` *`rights file`* changes the permissions for the file.

There are two ways of presenting rights. Among them, the symbolic representation is probably the easiest to understand and remember. It involves the letter symbols mentioned above. You can define rights for each category of users (u/g/o), by setting them explicitly (with =), by adding (+), or subtracting (-). Thus the u=rwx,g+rw,o-r formula gives the owner read, write, and execute rights, adds read and write rights for the owner group, and removes read rights for other users. Rights not altered by the addition or subtraction in such a command remain unmodified. The letter a, for "all", covers all three categories of users, so that a=rx grants all three categories the same rights (read and execute, but not write).

The (octal) numeric representation associates each right with a value: 4 for read, 2 for write, and 1 for execute. We associate each combination of rights with the sum of the figures. Each value is then assigned to different categories of users by putting them end to end in the usual order (owner, group, others).

For instance, the `chmod 754` *`file`* command will set the following rights: read, write and execute for the owner (since 7 = 4 + 2 + 1); read and execute for the group (since 5 = 4 + 1); read-only for others. The 0 means no rights; thus `chmod 600` *`file`* allows for read/write rights for the owner, and no rights for anyone else. The most frequent right combinations are 755 for executable files and directories, and 644 for data files.

To represent special rights, you can prefix a fourth digit to this number according to the same principle, where the setuid, setgid and sticky bits are 4, 2 and 1, respectively. `chmod 4754` will associate the setuid bit with the previously described rights.

Note that the use of octal notation only allows to set all the rights at once on a file; you cannot use it to simply add a new right, such as read access for the group owner, since you must take into account the existing rights and compute the new corresponding numerical value.

TIP

Recursive operation

Sometimes we have to change rights for an entire file tree. All the commands above have a `-R` option to operate recursively in sub-directories.

The distinction between directories and files sometimes causes problems with recursive operations. That is why the "X" letter has been introduced in the symbolic representation of rights. It represents a right to execute which applies only to directories (and not to files lacking this right). Thus, `chmod -R a+X` *`directory`* will only add execute rights for all categories of users (`a`) for all of the sub-directories and files for which at least one category of user (even if their sole owner) already has execute rights.

TIP

Changing the user and group

Frequently you want to change the group of a file at the same time that you change the owner. The `chown` command has a special syntax for that: `chown` *`user:group file`*

GOING FURTHER

`umask`

When an application creates a file, it assigns indicative permissions, knowing that the system automatically removes certain rights, given by the command `umask`. Enter `umask` in a shell; you will see a mask such as `0022`. This is simply an octal representation of the rights to be systematically removed (in this case, the write right for the group and other users).

If you give it a new octal value, the `umask` command modifies the mask. Used in a shell initialization file (for example, `~/.bash_profile`), it will effectively change the default mask for your work sessions.

9.4. Administration Interfaces

Using a graphical interface for administration is interesting in various circumstances. An administrator does not necessarily know all the configuration details for all their services, and doesn't always have the time to go seeking out the documentation on the matter. A graphical interface for administration can thus accelerate the deployment of a new service. It can also simplify the setup of services which are hard to configure.

Such an interface is only an aid, and not an end in itself. In all cases, the administrator must master its behavior in order to understand and work around any potential problem.

Since no interface is perfect, you may be tempted to try several solutions. This is to be avoided as much as possible, since different tools are sometimes incompatible in their work methods. Even if they all aim to be very flexible and try to adopt the configuration file as a single reference, they are not always able to integrate external changes.

9.4.1. Administrating on a Web Interface: `webmin`

This is, without a doubt, one of the most successful administration interfaces. It is a modular system managed through a web browser, covering a wide array of areas and tools. Furthermore, it is internationalized and available in many languages.

Sadly, webmin is no longer part of Debian. Its Debian maintainer — Jaldhar H. Vyas — removed the packages he created because he no longer had the time required to maintain them at an acceptable quality level. Nobody has officially taken over, so *Jessie* does not have the webmin package.

There is, however, an unofficial package distributed on the webmin.com website. Contrary to the original Debian packages, this package is monolithic; all of its configuration modules are installed and activated by default, even if the corresponding service is not installed on the machine.

SECURITY

Changing the root password

On the first login, identification is conducted with the root username and its usual password. It is recommended to change the password used for webmin as soon as possible, so that if it is compromised, the root password for the server will not be involved, even if this confers important administrative rights to the machine.

Beware Since webmin has so many features, a malicious user accessing it could compromise the security of the entire system. In general, interfaces of this kind are not recommended for important systems with strong security constraints (firewall, sensitive servers, etc.).

Webmin is used through a web interface, but it does not require Apache to be installed. Essentially, this software has its own integrated mini web server. This server listens by default on port 10000 and accepts secure HTTP connections.

Included modules cover a wide variety of services, among which:

- all base services: creation of users and groups, management of crontab files, init scripts, viewing of logs, etc.
- bind: DNS server configuration (name service);
- postfix: SMTP server configuration (e-mail);
- inetd: configuration of the inetd super-server;
- quota: user quota management;
- dhcpd: DHCP server configuration;
- proftpd: FTP server configuration;
- samba: Samba file server configuration;
- software: installation or removal of software from Debian packages and system updates.

The administration interface is available in a web browser at https://localhost:10000. Beware! Not all the modules are directly usable. Sometimes they must be configured by specifying the locations of the corresponding configuration files and some executable files (program). Frequently the system will politely prompt you when it fails to activate a requested module.

ALTERNATIVE

GNOME control center

The GNOME project also provides multiple administration interfaces that are usually accessible via the "Settings" entry in the user menu on the top right. `gnome-control-center` is the main program that brings them all together but many of the system wide configuration tools are effectively provided by other packages (*accountsservice*, *system-config-printer*, etc.). Although they are easy to use, these applications cover only a limited number of base services: user management, time configuration, network configuration, printer configuration, and so on.

9.4.2. Configuring Packages: `debconf`

Many packages are automatically configured after asking a few questions during installation through the Debconf tool. These packages can be reconfigured by running `dpkg-reconfigure` *package*.

For most cases, these settings are very simple; only a few important variables in the configuration file are changed. These variables are often grouped between two "demarcation" lines so that reconfiguration of the package only impacts the enclosed area. In other cases, reconfiguration will not change anything if the script detects a manual modification of the configuration file, in order to preserve these human interventions (because the script can't ensure that its own modifications will not disrupt the existing settings).

DEBIAN POLICY

Preserving changes

The Debian Policy expressly stipulates that everything should be done to preserve manual changes made to a configuration file, so more and more scripts take precautions when editing configuration files. The general principle is simple: the script will only make changes if it knows the status of the configuration file, which is verified by comparing the checksum of the file against that of the last automatically generated file. If they are the same, the script is authorized to change the configuration file. Otherwise, it determines that the file has been changed and asks what action it should take (install the new file, save the old file, or try to integrate the new changes with the existing file). This precautionary principle has long been unique to Debian, but other distributions have gradually begun to embrace it.

The `ucf` program (from the Debian package of the same name) can be used to implement such a behavior.

9.5. `syslog` System Events

9.5.1. Principle and Mechanism

The `rsyslogd` daemon is responsible for collecting service messages coming from applications and the kernel, then dispatching them into log files (usually stored in the `/var/log/` directory). It obeys the `/etc/rsyslog.conf` configuration file.

Each log message is associated with an application subsystem (called "facility" in the documentation):

- auth and authpriv: for authentication;
- cron: comes from task scheduling services, `cron` and `atd`;
- daemon: affects a daemon without any special classification (DNS, NTP, etc.);
- ftp: concerns the FTP server;
- kern: message coming from the kernel;
- lpr: comes from the printing subsystem;
- mail: comes from the e-mail subsystem;
- news: Usenet subsystem message (especially from an NNTP — Network News Transfer Protocol — server that manages newsgroups);
- syslog: messages from the `syslogd` server, itself;
- user: user messages (generic);
- uucp: messages from the UUCP server (Unix to Unix Copy Program, an old protocol notably used to distribute e-mail messages);
- local0 to local7: reserved for local use.

Each message is also associated with a priority level. Here is the list in decreasing order:

- emerg: "Help!" There is an emergency, the system is probably unusable.
- alert: hurry up, any delay can be dangerous, action must be taken immediately;
- crit: conditions are critical;
- err: error;
- warn: warning (potential error);
- notice: conditions are normal, but the message is important;
- info: informative message;
- debug: debugging message.

9.5.2. The Configuration File

The syntax of the `/etc/rsyslog.conf` file is detailed in the `rsyslog.conf(5)` manual page, but there is also HTML documentation available in the *rsyslog-doc* package (`/usr/share/doc/rsyslog-doc/html/index.html`). The overall principle is to write "selector" and "action" pairs. The selector defines all relevant messages, and the actions describes how to deal with them.

Syntax of the Selector

The selector is a semicolon-separated list of *subsystem.priority* pairs (example: auth.notice; mail.info). An asterisk may represent all subsystems or all priorities (examples: *.alert or mail.*). Several subsystems can be grouped, by separating them with a comma (example: auth,mail.

info). The priority indicated also covers messages of equal or higher priority; thus auth.alert indicates the auth subsystem messages of alert or emerg priority. Prefixed with an exclamation point (!), it indicates the opposite, in other words the strictly lower priorities; auth.!notice, thus, indicates messages issued from auth, with info or debug priority. Prefixed with an equal sign (=), it corresponds to precisely and only the priority indicated (auth.=notice only concerns messages from auth with notice priority).

Each element in the list on the selector overrides previous elements. It is thus possible to restrict a set or to exclude certain elements from it. For example, kern.info;kern.!err means messages from the kernel with priority between info and warn. The none priority indicates the empty set (no priorities), and may serve to exclude a subsystem from a set of messages. Thus, *.crit; kern.none indicates all the messages of priority equal to or higher than crit not coming from the kernel.

Syntax of Actions

BACK TO BASICS

The named pipe, a persistent pipe

A named pipe is a particular type of file that operates like a traditional pipe (the pipe that you make with the "|" symbol on the command line), but via a file. This mechanism has the advantage of being able to relate two unrelated processes. Anything written to a named pipe blocks the process that writes until another process attempts to read the data written. This second process reads the data written by the first, which can then resume execution.

Such a file is created with the `mkfifo` command.

The various possible actions are:

- add the message to a file (example: `/var/log/messages`);
- send the message to a remote `syslog` server (example: @log.falcot.com);
- send the message to an existing named pipe (example: |/dev/xconsole);
- send the message to one or more users, if they are logged in (example: root,rhertzog);
- send the message to all logged in users (example: *);
- write the message in a text console (example: /dev/tty8).

SECURITY

Forwarding logs

It is a good idea to record the most important logs on a separate machine (perhaps dedicated for this purpose), since this will prevent any possible intruder from removing traces of their intrusion (unless, of course, they also compromise this other server). Furthermore, in the event of a major problem (such as a kernel crash), you have the logs available on another machine, which increases your chances of determining the sequence of events that caused the crash.

To accept log messages sent by other machines, you must reconfigure *rsyslog*: in practice, it is sufficient to activate the ready-for-use entries in `/etc/rsyslog.conf` (`$ModLoad imudp` and `$UDPServerRun 514`).

9.6. The `inetd` Super-Server

Inetd (often called "Internet super-server") is a server of servers. It executes rarely used servers on demand, so that they do not have to run continuously.

The `/etc/inetd.conf` file lists these servers and their usual ports. The `inetd` command listens to all of them; when it detects a connection to any such port, it executes the corresponding server program.

DEBIAN POLICY

Register a server in `inetd.conf`

Packages frequently want to register a new server in the `/etc/inetd.conf` file, but Debian Policy prohibits any package from modifying a configuration file that it doesn't own. This is why the `update-inetd` script (in the package with the same name) was created: It manages the configuration file, and other packages can thus use it to register a new server to the super-server's configuration.

Each significant line of the `/etc/inetd.conf` file describes a server through seven fields (separated by spaces):

- The TCP or UDP port number, or the service name (which is mapped to a standard port number with the information contained in the `/etc/services` file).
- The socket type: stream for a TCP connection, dgram for UDP datagrams.
- The protocol: tcp or udp.
- The options: two possible values: wait or nowait, to tell `inetd` whether it should wait or not for the end of the launched process before accepting another connection. For TCP connections, easily multiplexable, you can usually use nowait. For programs responding over UDP, you should use nowait only if the server is capable of managing several connections in parallel. You can suffix this field with a period, followed by the maximum number of connections authorized per minute (the default limit is 256).
- The user name of the user under whose identity the server will run.
- The full path to the server program to execute.
- The arguments: this is a complete list of the program's arguments, including its own name (argv[0] in C).

The following example illustrates the most common cases:

Example 9.1 *Excerpt from `/etc/inetd.conf`*

```
talk   dgram  udp wait    nobody.tty /usr/sbin/in.talkd in.talkd
finger stream tcp nowait  nobody     /usr/sbin/tcpd     in.fingerd
ident  stream tcp nowait  nobody     /usr/sbin/identd   identd -i
```

The `tcpd` program is frequently used in the `/etc/inetd.conf` file. It allows limiting incoming connections by applying access control rules, documented in the `hosts_access(5)` manual

page, and which are configured in the `/etc/hosts.allow` and `/etc/hosts.deny` files. Once it has been determined that the connection is authorized, `tcpd` executes the real server (like `in.fingerd` in our example). It is worth noting that `tcpd` relies on the name under which it was invoked (that is the first argument, argv[0]) to identify the real program to run. So you should not start the arguments list with tcpd but with the program that must be wrapped.

COMMUNITY

Wietse Venema

Wietse Venema, whose expertise in security has made him a renowned programmer, is the author of the `tcpd` program. He is also the main creator of Postfix, the modular e-mail server (SMTP, Simple Mail Transfer Protocol), designed to be safer and more reliable than `sendmail`, which features a long history of security vulnerabilities.

ALTERNATIVE

Other `inetd` commands

While Debian installs *openbsd-inetd* by default, there is no lack of alternatives: we can mention *inetutils-inetd*, *micro-inetd*, *rlinetd* and *xinetd*.

This last incarnation of a super-server offers very interesting possibilities. Most notably, its configuration can be split into several files (stored, of course, in the `/etc/xinetd.d/` directory), which can make an administrator's life easier.

Last but not least, it is even possible to emulate `inetd`'s behaviour with `systemd`'s socket-activation mechanism (see section 9.1.1, "The systemd init system" page 182).

9.7. Scheduling Tasks with `cron` and `atd`

`cron` is the daemon responsible for executing scheduled and recurring commands (every day, every week, etc.); `atd` is that which deals with commands to be executed a single time, but at a specific moment in the future.

In a Unix system, many tasks are scheduled for regular execution:

- rotating the logs;
- updating the database for the `locate` program;
- back-ups;
- maintenance scripts (such as cleaning out temporary files).

By default, all users can schedule the execution of tasks. Each user has thus their own *crontab* in which they can record scheduled commands. It can be edited by running `crontab -e` (its content is stored in the `/var/spool/cron/crontabs/`*user* file).

SECURITY

Restricting `cron` or `atd`

You can restrict access to `cron` by creating an explicit authorization file (whitelist) in `/etc/cron.allow`, in which you indicate the only users authorized to schedule commands. All others will automatically be deprived of this feature. Conversely, to only block one or two troublemakers, you could write their username in the explicit prohibition file (blacklist), `/etc/cron.deny`. This same feature is available for `atd`, with the `/etc/at.allow` and `/etc/at.deny` files.

The root user has their own *crontab*, but can also use the `/etc/crontab` file, or write additional *crontab* files in the `/etc/cron.d` directory. These last two solutions have the advantage of being able to specify the user identity to use when executing the command.

The *cron* package includes by default some scheduled commands that execute:

- programs in the `/etc/cron.hourly/` directory once per hour;
- programs in `/etc/cron.daily/` once per day;
- programs in `/etc/cron.weekly/` once per week;
- programs in `/etc/cron.monthly/` once per month.

Many Debian packages rely on this service: by putting maintenance scripts in these directories, they ensure optimal operation of their services.

9.7.1. Format of a `crontab` File

TIP

Text shortcuts for `cron`

`cron` recognizes some abbreviations which replace the first five fields in a `crontab` entry. They correspond to the most classic scheduling options:

- `@yearly`: once per year (January 1, at 00:00);
- `@monthly`: once per month (the 1st of the month, at 00:00);
- `@weekly`: once per week (Sunday at 00:00);
- `@daily`: once per day (at 00:00);
- `@hourly`: once per hour (at the beginning of each hour).

SPECIAL CASE

`cron` and daylight savings time

In Debian, `cron` takes the time change (for Daylight Savings Time, or in fact for any significant change in the local time) into account as best as it can. Thus, the commands that should have been executed during an hour that never existed (for example, tasks scheduled at 2:30 am during the Spring time change in France, since at 2:00 am the clock jumps directly to 3:00 am) are executed shortly after the time change (thus around 3:00 am DST). On the other hand, in autumn, when commands would be executed several times (2:30 am DST, then an hour later at 2:30 am standard time, since at 3:00 am DST the clock turns back to 2:00 am) are only executed once.

Be careful, however, if the order in which the different scheduled tasks and the delay between their respective executions matters, you should check the compatibility of these constraints with `cron`'s behavior; if necessary, you can prepare a special schedule for the two problematic nights per year.

Each significant line of a *crontab* describes a scheduled command with the six (or seven) following fields:

- the value for the minute (number from 0 to 59);
- the value for the hour (from 0 to 23);

- the value for the day of the month (from 1 to 31);
- the value for the month (from 1 to 12);
- the value for the day of the week (from 0 to 7, 1 corresponding to Monday, Sunday being represented by both 0 and 7; it is also possible to use the first three letters of the name of the day of the week in English, such as Sun, Mon, etc.);
- the user name under whose identity the command must be executed (in the `/etc/crontab` file and in the fragments located in `/etc/cron.d/`, but not in the users' own crontab files);
- the command to execute (when the conditions defined by the first five columns are met).

All these details are documented in the `crontab(5)` man page.

Each value can be expressed in the form of a list of possible values (separated by commas). The syntax a-b describes the interval of all the values between a and b. The syntax a-b/c describes the interval with an increment of c (example: 0-10/2 means 0,2,4,6,8,10). An asterisk * is a wildcard, representing all possible values.

Example 9.2 *Sample `crontab` file*

```
#Format
#min hour day mon dow  command

# Download data every night at 7:25 pm
 25  19   *   *   *    $HOME/bin/get.pl

# 8:00 am, on weekdays (Monday through Friday)
 00  08   *   *   1-5  $HOME/bin/dosomething

# Restart the IRC proxy after each reboot
@reboot /usr/bin/dircproxy
```

TIP

Executing a command on boot

To execute a command a single time, just after booting the computer, you can use the `@reboot` macro (a simple restart of `cron` does not trigger a command scheduled with `@reboot`). This macro replaces the first five fields of an entry in the *crontab*.

ALTERNATIVE

Emulating `cron` with `systemd`

It is possible to emulate part of `cron`'s behaviour with `systemd`'s timer mechanism (see section 9.1.1, "The systemd init system" page 182).

9.7.2. Using the `at` Command

The `at` executes a command at a specified moment in the future. It takes the desired time and date as command-line parameters, and the command to be executed in its standard input. The

command will be executed as if it had been entered in the current shell. `at` even takes care to retain the current environment, in order to reproduce the same conditions when it executes the command. The time is indicated by following the usual conventions: 16:12 or 4:12pm represents 4:12 pm. The date can be specified in several European and Western formats, including DD.MM.YY (27.07.15 thus representing 27 July 2015), YYYY-MM-DD (this same date being expressed as 2015-07-27), MM/DD/[CC]YY (ie., 12/25/15 or 12/25/2015 will be December 25, 2015), or simple MMDD[CC]YY (so that 122515 or 12252015 will, likewise, represent December 25, 2015). Without it, the command will be executed as soon as the clock reaches the time indicated (the same day, or tomorrow if that time has already passed on the same day). You can also simply write "today" or "tomorrow", which is self-explanatory.

```
$ at 09:00 27.07.15 <<END
> echo "Don't forget to wish a Happy Birthday to Raphaël!" \
>   | mail lolando@debian.org
> END
warning: commands will be executed using /bin/sh
job 31 at Mon Jul 27 09:00:00 2015
```

An alternative syntax postpones the execution for a given duration: `at now + number period`. The *period* can be minutes, hours, days, or weeks. The *number* simply indicates the number of said units that must elapse before execution of the command.

To cancel a task scheduled by `cron`, simply run `crontab -e` and delete the corresponding line in the *crontab* file. For `at` tasks, it is almost as easy: run `atrm task-number`. The task number is indicated by the `at` command when you scheduled it, but you can find it again with the `atq` command, which gives the current list of scheduled tasks.

9.8. Scheduling Asynchronous Tasks: `anacron`

`anacron` is the daemon that completes `cron` for computers that are not on at all times. Since regular tasks are usually scheduled for the middle of the night, they will never be executed if the computer is off at that time. The purpose of `anacron` is to execute them, taking into account periods in which the computer is not working.

Please note that `anacron` will frequently execute such activity a few minutes after booting the machine, which can render the computer less responsive. This is why the tasks in the `/etc/anacrontab` file are started with the `nice` command, which reduces their execution priority and thus limits their impact on the rest of the system. Beware, the format of this file is not the same as that of `/etc/crontab`; if you have particular needs for `anacron`, see the `anacrontab(5)` manual page.

Installation of the *anacron* package deactivates execution by `cron` of the scripts in the `/etc/cron.hourly/`, `/etc/cron.daily/`, `/etc/cron.weekly/`, and `/etc/cron.monthly/` directories. This avoids their double execution by `anacron` and `cron`. The `cron` command remains active and will continue to handle the other scheduled tasks (especially those scheduled by users).

BACK TO BASICS

Priorities and `nice`

Unix systems (and thus Linux) are multi-tasking and multi-user systems. Indeed, several processes can run in parallel, and be owned by different users: the kernel mediates access to the resources between the different processes. As a part of this task, it has a concept of priority, which allows it to favor certain processes over others, as needed. When you know that a process can run in low priority, you can indicate so by running it with `nice` *`program`*. The program will then have a smaller share of the CPU, and will have a smaller impact on other running processes. Of course, if no other processes needs to run, the program will not be artificially held back.

`nice` works with levels of "niceness": the positive levels (from 1 to 19) progressively lower the priority, while the negative levels (from -1 to -20) will increase it — but only root can use these negative levels. Unless otherwise indicated (see the `nice(1)` manual page), `nice` increases the current level by 10.

If you discover that an already running task should have been started with `nice` it is not too late to fix it; the `renice` command changes the priority of an already running process, in either direction (but reducing the "niceness" of a process is reserved for the root user).

9.9. Quotas

The quota system allows limiting disk space allocated to a user or group of users. To set it up, you must have a kernel that supports it (compiled with the `CONFIG_QUOTA` option) — as is the case with Debian kernels. The quota management software is found in the *quota* Debian package.

To activate quota in a filesystem, you have to indicate the usrquota and grpquota options in `/etc/fstab` for the user and group quotas, respectively. Rebooting the computer will then update the quotas in the absence of disk activity (a necessary condition for proper accounting of already used disk space).

The `edquota` *`user`* (or `edquota -g` *`group`*) command allows you to change the limits while examining current disk space usage.

GOING FURTHER

Defining quotas with a script

The `setquota` program can be used in a script to automatically change many quotas. Its `setquota(8)` manual page details the syntax to use.

The quota system allows you to set four limits:

- two limits (called "soft" and "hard") refer to the number of blocks consumed. If the filesystem was created with a block-size of 1 kibibyte, a block contains 1024 bytes from the same file. Unsaturated blocks thus induce losses of disk space. A quota of 100 blocks, which theoretically allows storage of 102,400 bytes, will however be saturated with just 100 files of 500 bytes each, only representing 50,000 bytes in total.
- two limits (soft and hard) refer to the number of inodes used. Each file occupies at least one inode to store information about it (permissions, owner, timestamp of last access, etc.). It is thus a limit on the number of user files.

A "soft" limit can be temporarily exceeded; the user will simply be warned that they are exceeding the quota by the `warnquota` command, which is usually invoked by `cron`. A "hard" limit can never be exceeded: the system will refuse any operation that will cause a hard quota to be exceeded.

VOCABULARY

Blocks and inodes

The filesystem divides the hard drive into blocks — small contiguous areas. The size of these blocks is defined during creation of the filesystem, and generally varies between 1 and 8 kibibytes.

A block can be used either to store the real data of a file, or for meta-data used by the filesystem. Among this meta-data, you will especially find the inodes. An inode uses a block on the hard drive (but this block is not taken into consideration in the block quota, only in the inode quota), and contains both the information on the file to which it corresponds (name, owner, permissions, etc.) and the pointers to the data blocks that are actually used. For very large files that occupy more blocks than it is possible to reference in a single inode, there is an indirect block system; the inode references a list of blocks that do not directly contain data, but another list of blocks.

With the `edquota -t` command, you can define a maximum authorized "grace period" within which a soft limit may be exceeded. After this period, the soft limit will be treated like a hard limit, and the user will have to reduce their disk space usage to within this limit in order to be able to write anything to the hard drive.

GOING FURTHER

Setting up a default quota for new users

To automatically setup a quota for new users, you have to configure a template user (with `edquota` or `setquota`) and indicate their user name in the `QUOTAUSER` variable in the `/etc/adduser.conf` file. This quota configuration will then be automatically applied to each new user created with the `adduser` command.

9.10. Backup

Making backups is one of the main responsibilities of any administrator, but it is a complex subject, involving powerful tools which are often difficult to master.

Many programs exist, such as `amanda`, `bacula`, `BackupPC`. Those are client/server system featuring many options, whose configuration is rather difficult. Some of them provide user-friendly web interfaces to mitigate this. But Debian contains dozens of other backup software covering all possible use cases, as you can easily confirm with `apt-cache search backup`.

Rather than detailing some of them, this section will present the thoughts of the Falcot Corp administrators when they defined their backup strategy.

At Falcot Corp, backups have two goals: recovering erroneously deleted files, and quickly restoring any computer (server or desktop) whose hard drive has failed.

9.10.1. Backing Up with rsync

Backups on tape having been deemed too slow and costly, data will be backed up on hard drives on a dedicated server, on which the use of software RAID (see section 12.1.1, "Software RAID" page 302) will protect the data from hard drive failure. Desktop computers are not backed up individually, but users are advised that their personal account on their department's file server will be backed up. The `rsync` command (from the package of the same name) is used daily to back up these different servers.

BACK TO BASICS

The hard link, a second name for the file

A hard link, as opposed to a symbolic link, cannot be differentiated from the linked file. Creating a hard link is essentially the same as giving an existing file a second name. This is why the deletion of a hard link only removes one of the names associated with the file. As long as another name is still assigned to the file, the data therein remain present on the filesystem. It is interesting to note that, unlike a copy, the hard link does not take up additional space on the hard drive.

A hard link is created with the `ln` *`target link`* command. The *link* file is then a new name for the *target* file. Hard links can only be created on the same filesystem, while symbolic links are not subject to this limitation.

The available hard drive space prohibits implementation of a complete daily backup. As such, the `rsync` command is preceded by a duplication of the content of the previous backup with hard links, which prevents usage of too much hard drive space. The `rsync` process then only replaces files that have been modified since the last backup. With this mechanism a great number of backups can be kept in a small amount of space. Since all backups are immediately available and accessible (for example, in different directories of a given share on the network), you can quickly make comparisons between two given dates.

This backup mechanism is easily implemented with the `dirvish` program. It uses a backup storage space ("bank" in its vocabulary) in which it places timestamped copies of sets of backup files (these sets are called "vaults" in the dirvish documentation).

The main configuration is in the `/etc/dirvish/master.conf` file. It defines the location of the backup storage space, the list of "vaults" to manage, and default values for expiration of the backups. The rest of the configuration is located in the *`bank`*`/`*`vault`*`/dirvish/default.conf` files and contains the specific configuration for the corresponding set of files.

Example 9.3 *The `/etc/dirvish/master.conf` file*

```
bank:
    /backup
exclude:
    lost+found/
    core
    *~
Runall:
    root    22:00
```

```
expire-default: +15 days
expire-rule:
#   MIN HR    DOM MON       DOW  STRFTIME_FMT
    *   *     *   *         1    +3 months
    *   *     1-7 *         1    +1 year
    *   *     1-7 1,4,7,10  1
```

The bank setting indicates the directory in which the backups are stored. The exclude setting allows you to indicate files (or file types) to exclude from the backup. The Runall is a list of file sets to backup with a time-stamp for each set, which allows you to assign the correct date to the copy, in case the backup is not triggered at precisely the assigned time. You have to indicate a time just before the actual execution time (which is, by default, 10:04 pm in Debian, according to `/etc/cron.d/dirvish`). Finally, the expire-default and expire-rule settings define the expiration policy for backups. The above example keeps forever backups that are generated on the first Sunday of each quarter, deletes after one year those from the first Sunday of each month, and after 3 months those from other Sundays. Other daily backups are kept for 15 days. The order of the rules does matter, Dirvish uses the last matching rule, or the expire-default one if no other expire-rule matches.

IN PRACTICE

Scheduled expiration

The expiration rules are not used by `dirvish-expire` to do its job. In reality, the expiration rules are applied when creating a new backup copy to define the expiration date associated with that copy. `dirvish-expire` simply peruses the stored copies and deletes those for which the expiration date has passed.

Example 9.4 *The `/backup/root/dirvish/default.conf` file*

```
client: rivendell.falcot.com
tree: /
xdev: 1
index: gzip
image-default: %Y%m%d
exclude:
    /var/cache/apt/archives/*.deb
    /var/cache/man/**
    /tmp/**
    /var/tmp/**
    *.bak
```

The above example specifies the set of files to back up: these are files on the machine *rivendell.falcot.com* (for local data backup, simply specify the name of the local machine as indicated by `hostname`), especially those in the root tree (tree:/), except those listed in exclude. The backup will be limited to the contents of one filesystem (xdev:1). It will not include files from other mount points. An index of saved files will be generated (index:gzip), and the image will be named according to the current date (image-default:%Y%m%d).

There are many options available, all documented in the `dirvish.conf(5)` manual page. Once these configuration files are setup, you have to initialize each file set with the `dirvish --vault` *`vault`* `--init` command. From there on the daily invocation of `dirvish-runall` will automatically create a new backup copy just after having deleted those that expired.

IN PRACTICE

Remote backup over SSH

When dirvish needs to save data to a remote machine, it will use `ssh` to connect to it, and will start `rsync` as a server. This requires the root user to be able to automatically connect to it. The use of an SSH authentication key allows precisely that (see section 9.2.1.1, "Key-Based Authentication" page 192).

9.10.2. Restoring Machines without Backups

Desktop computers, which are not backed up, will be easy to reinstall from custom DVD-ROMs prepared with *Simple-CDD* (see section 12.3.3, "Simple-CDD: The All-In-One Solution" page 343). Since this performs an installation from scratch, it loses any customization that can have been made after the initial installation. This is fine since the systems are all hooked to a central LDAP directory for accounts and most desktop applications are preconfigured thanks to dconf (see section 13.3.1, "GNOME" page 359 for more information about this).

The Falcot Corp administrators are aware of the limits in their backup policy. Since they can't protect the backup server as well as a tape in a fireproof safe, they have installed it in a separate room so that a disaster such as a fire in the server room won't destroy backups along with everything else. Furthermore, they do an incremental backup on DVD-ROM once per week — only files that have been modified since the last backup are included.

GOING FURTHER

Backing up SQL and LDAP services

Many services (such as SQL or LDAP databases) cannot be backed up by simply copying their files (unless they are properly interrupted during creation of the backups, which is frequently problematic, since they are intended to be available at all times). As such, it is necessary to use an "export" mechanism to create a "data dump" that can be safely backed up. These are often quite large, but they compress well. To reduce the storage space required, you will only store a complete text file per week, and a `diff` each day, which is created with a command of the type `diff` *`file_from_yesterday file_from_today`*. The `xdelta` program produces incremental differences from binary dumps.

CULTURE

***TAR*, the standard for tape backups**

Historically, the simplest means of making a backup on Unix was to store a *TAR* archive on a tape. The `tar` command even got its name from "Tape ARchive".

9.11. Hot Plugging: *hotplug*

9.11.1. Introduction

The *hotplug* kernel subsystem dynamically handles the addition and removal of devices, by loading the appropriate drivers and by creating the corresponding device files (with the help of

udevd). With modern hardware and virtualization, almost everything can be hotplugged: from the usual USB/PCMCIA/IEEE 1394 peripherals to SATA hard drives, but also the CPU and the memory.

The kernel has a database that associates each device ID with the required driver. This database is used during boot to load all the drivers for the peripheral devices detected on the different buses, but also when an additional hotplug device is connected. Once the device is ready for use, a message is sent to `udevd` so it will be able to create the corresponding entry in `/dev/`.

9.11.2. The Naming Problem

Before the appearance of hotplug connections, it was easy to assign a fixed name to a device. It was based simply on the position of the devices on their respective bus. But this is not possible when such devices can come and go on the bus. The typical case is the use of a digital camera and a USB key, both of which appear to the computer as disk drives. The first one connected may be `/dev/sdb` and the second `/dev/sdc` (with `/dev/sda` representing the computer's own hard drive). The device name is not fixed; it depends on the order in which devices are connected.

Additionally, more and more drivers use dynamic values for devices' major/minor numbers, which makes it impossible to have static entries for the given devices, since these essential characteristics may vary after a reboot.

udev was created precisely to solve this problem.

IN PRACTICE

Network card management

Many computers have multiple network cards (sometimes two wired interfaces and a wifi interface), and with *hotplug* support on most bus types, the Linux kernel does not guarantee fixed naming of network interfaces. But users who want to configure their network in `/etc/network/interfaces` need a fixed name.

It would be difficult to ask every user to create their own *udev* rules to address this problem. This is why *udev* was configured in a rather peculiar manner; on first boot (and, more generally, each time that a new network card appears) it uses the name of the network interface and its MAC address to create new rules that will reassign the same name on subsequent boots. These rules are stored in `/etc/udev/rules.d/70-persistent-net.rules`.

This mechanism has some side effects that you should know about. Let's consider the case of a computer that has only one PCI network card. The network interface is named `eth0`, logically. Now say the card breaks down, and the administrator replaces it; the new card will have a new MAC address. Since the old card was assigned the name, `eth0`, the new one will be assigned `eth1`, even though the `eth0` card is gone for good (and the network will not be functional because `/etc/network/interfaces` likely configures an `eth0` interface). In this case, it is enough to simply delete the `/etc/udev/rules.d/70-persistent-net.rules` file before rebooting the computer. The new card will then be given the expected `eth0` name.

9.11.3. How *udev* Works

When *udev* is notified by the kernel of the appearance of a new device, it collects various information on the given device by consulting the corresponding entries in `/sys/`, especially those that uniquely identify it (MAC address for a network card, serial number for some USB devices, etc.).

Armed with all of this information, *udev* then consults all of the rules contained in `/etc/udev/rules.d/` and `/lib/udev/rules.d/`. In this process it decides how to name the device, what symbolic links to create (to give it alternative names), and what commands to execute. All of these files are consulted, and the rules are all evaluated sequentially (except when a file uses "GOTO" directives). Thus, there may be several rules that correspond to a given event.

The syntax of rules files is quite simple: each row contains selection criteria and variable assignments. The former are used to select events for which there is a need to react, and the latter defines the action to take. They are all simply separated with commas, and the operator implicitly differentiates between a selection criterion (with comparison operators, such as == or ⊠) or an assignment directive (with operators such as =, += or :=).

Comparison operators are used on the following variables:

- KERNEL: the name that the kernel assigns to the device;
- ACTION: the action corresponding to the event ("add" when a device has been added, "remove" when it has been removed);
- DEVPATH: the path of the device's `/sys/` entry;
- SUBSYSTEM: the kernel subsystem which generated the request (there are many, but a few examples are "usb", "ide", "net", "firmware", etc.);
- ATTR{*attribute*}: file contents of the *attribute* file in the `/sys/$devpath/` directory of the device. This is where you find the MAC address and other bus specific identifiers;
- KERNELS, SUBSYSTEMS and ATTRS{*attributes*} are variations that will try to match the different options on one of the parent devices of the current device;
- PROGRAM: delegates the test to the indicated program (true if it returns 0, false if not). The content of the program's standard output is stored so that it can be reused by the RESULT test;
- RESULT: execute tests on the standard output stored during the last call to PROGRAM.

The right operands can use pattern expressions to match several values at the same time. For instance, * matches any string (even an empty one); ? matches any character, and [] matches the set of characters listed between the square brackets (or the opposite thereof if the first character is an exclamation point, and contiguous ranges of characters are indicated like a-z).

Regarding the assignment operators, = assigns a value (and replaces the current value); in the case of a list, it is emptied and contains only the value assigned. := does the same, but prevents later changes to the same variable. As for +=, it adds an item to a list. The following variables can be changed:

- NAME: the device filename to be created in `/dev/`. Only the first assignment counts; the others are ignored;
- SYMLINK: the list of symbolic links that will point to the same device;
- OWNER, GROUP and MODE define the user and group that owns the device, as well as the associated permission;
- RUN: the list of programs to execute in response to this event.

The values assigned to these variables may use a number of substitutions:

- $kernel or %k: equivalent to KERNEL;
- $number or %n: the order number of the device, for example, for sda3, it would be "3";
- $devpath or %p: equivalent to DEVPATH;
- $attr{*attribute*} or %s{*attribute*}: equivalent to ATTRS{*attribute*};
- $major or %M: the kernel major number of the device;
- $minor or %m: the kernel minor number of the device;
- $result or %c: the string output by the last program invoked by PROGRAM;
- and, finally, %% and $$ for the percent and dollar sign, respectively.

The above lists are not complete (they include only the most important parameters), but the `udev(7)` manual page should be exhaustive.

9.11.4. A concrete example

Let us consider the case of a simple USB key and try to assign it a fixed name. First, you must find the elements that will identify it in a unique manner. For this, plug it in and run `udevadm info -a -n /dev/sdc` (replacing */dev/sdc* with the actual name assigned to the key).

```
# udevadm info -a -n /dev/sdc
[...]
  looking at device '/devices/pci0000:00/0000:00:10.3/usb1/1-2/1-2.2/1-2.2:1.0/host9/
        target9:0:0/9:0:0:0/block/sdc':
    KERNEL=="sdc"
    SUBSYSTEM=="block"
    DRIVER==""
    ATTR{range}=="16"
    ATTR{ext_range}=="256"
    ATTR{removable}=="1"
    ATTR{ro}=="0"
    ATTR{size}=="126976"
    ATTR{alignment_offset}=="0"
    ATTR{capability}=="53"
    ATTR{stat}==" 51 100 1208 256 0 0 0
                  0 0 192 25 6"
    ATTR{inflight}==" 0 0"
```

```
[...]
  looking at parent device '/devices/pci0000:00/0000:00:10.3/usb1
        /1-2/1-2.2/1-2.2:1.0/host9/target9:0:0/9:0:0:0':
    KERNELS=="9:0:0:0"
    SUBSYSTEMS=="scsi"
    DRIVERS=="sd"
    ATTRS{device_blocked}=="0"
    ATTRS{type}=="0"
    ATTRS{scsi_level}=="3"
    ATTRS{vendor}=="I0MEGA  "
    ATTRS{model}=="UMni64MB*IOM2C4 "
    ATTRS{rev}=="    "
    ATTRS{state}=="running"
[...]
    ATTRS{max_sectors}=="240"
[...]
  looking at parent device '/devices/pci0000:00/0000:00:10.3/usb1/1-2/1-2.2':
    KERNELS=="9:0:0:0"
    SUBSYSTEMS=="usb"
    DRIVERS=="usb"
    ATTRS{configuration}=="iCfg"
    ATTRS{bNumInterfaces}==" 1"
    ATTRS{bConfigurationValue}=="1"
    ATTRS{bmAttributes}=="80"
    ATTRS{bMaxPower}=="100mA"
    ATTRS{urbnum}=="398"
    ATTRS{idVendor}=="4146"
    ATTRS{idProduct}=="4146"
    ATTRS{bcdDevice}=="0100"
[...]
    ATTRS{manufacturer}=="USB Disk"
    ATTRS{product}=="USB Mass Storage Device"
    ATTRS{serial}=="M004021000001"
[...]
```

To create a new rule, you can use tests on the device's variables, as well as those of one of the parent devices. The above case allows us to create two rules like these:

```
KERNEL=="sd?", SUBSYSTEM=="block", ATTRS{serial}=="M004021000001", SYMLINK+="usb_key/
       disk"
KERNEL=="sd?[0-9]", SUBSYSTEM=="block", ATTRS{serial}=="M004021000001", SYMLINK+="
       usb_key/part%n"
```

Once these rules are set in a file, named for example `/etc/udev/rules.d/010_local.rules`, you can simply remove and reconnect the USB key. You can then see that `/dev/usb_key/disk` represents the disk associated with the USB key, and `/dev/usb_key/part1` is its first partition.

GOING FURTHER

Debugging *udev*'s configuration

Like many daemons, `udevd` stores logs in `/var/log/daemon.log`. But it is not very verbose by default, and it is usually not enough to understand what is happening. The `udevadm control --log-priority=info` command increases the verbosity level and solves this problem. `udevadm control --log-priority=err` returns to the default verbosity level.

9.12. Power Management: Advanced Configuration and Power Interface (ACPI)

The topic of power management is often problematic. Indeed, properly suspending the computer requires that all the computer's device drivers know how to put them to standby, and that they properly reconfigure the devices upon waking. Unfortunately, there are still a few devices unable to sleep well under Linux, because their manufacturers have not provided the required specifications.

Linux supports ACPI (Advanced Configuration and Power Interface) — the most recent standard in power management. The *acpid* package provides a daemon that looks for power management related events (switching between AC and battery power on a laptop, etc.) and that can execute various commands in response.

BEWARE

Graphics card and standby

The graphics card driver is often the culprit when standby doesn't work properly. In that case, it is a good idea to test the latest version of the X.org graphics server.

After this overview of basic services common to many Unix systems, we will focus on the environment of the administered machines: the network. Many services are required for the network to work properly. They will be discussed in the next chapter.

Keywords

Network
Gateway
TCP/IP
IPv6
DNS
Bind
DHCP
QoS