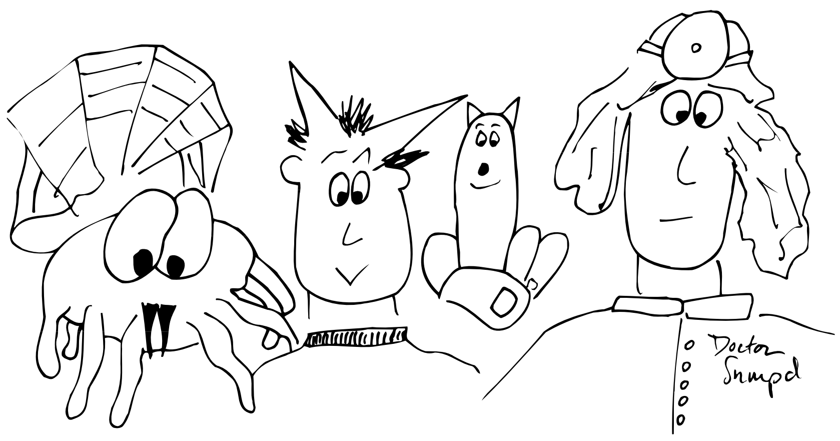
29 Daemons



A daemon is a background process that performs a specific function or system task. In keeping with the UNIX and Linux philosophy of modularity, daemons are programs rather than parts of the kernel. Many daemons start at boot time and continue to run as long as the system is up. Other daemons are started when needed and run only as long as they are useful.

“Daemon” was first used as a computer term by Mick Bailey, a British gentleman who was working on the CTSS programming staff at MIT during the early 1960s.[1] Mick quoted the Oxford English Dictionary in support of both the meaning and the spelling of the word. The words “daemon” and “demon” both come from the same root, but “daemon” is an older form and its meaning is somewhat different. A daemon is an attendant spirit that influences one’s character or personality. Daemons are not minions of evil *or* good; they’re creatures of independent thought and will. Daemons made their way from CTSS to Multics to UNIX to Linux, where they are so popular that they need a superdaemon (**xinetd** or **inetd**) to manage them.

This chapter presents a brief overview of the most common Linux daemons. Not all the daemons listed here are supplied with all Linux distributions, and not every daemon supplied with some Linux distribution is listed here. Besides making you more aware of how Linux works, a knowledge of what all the various daemons do will make you look really smart when one of your users asks, “What does **klogd** do?”

Before **inetd** was written, all daemons started at boot time and ran continuously (or more accurately, they blocked waiting for work to do). Over time, more and more

1. This bit of history comes from Jerry Saltzer at MIT, via Dennis Ritchie.

daemons were added to the system. The daemon population became so large that it began to cause performance problems. In response, the Berkeley gurus developed **inetd**, a daemon that starts other daemons as they are needed. **inetd** successfully popularized this superdaemon model, which remains a common way to minimize the number of processes running on a server. Most versions of UNIX and Linux now use a combination of **inetd** and always-running daemons.

There are many daemons that system administrators should be intimately familiar with, either because they require a lot of administration or because they play a large role in the day-to-day operation of the system. Some daemons that are described here in one or two lines have an entire chapter devoted to them elsewhere in this book. We provide cross-references where appropriate.

We start this chapter by introducing a couple of very important system daemons (**init** and **cron**) and then move on to a discussion of **xinetd** and **inetd**. Finally, we briefly describe most of the daemons a system administrator is likely to wrestle with on our four example distributions.

29.1 INIT: THE PRIMORDIAL PROCESS

init is the first process to run after the system boots, and in many ways it is the most important daemon. It always has a PID of 1 and is an ancestor of all user processes and all but a few system processes.

At startup, **init** either places the system in single-user mode or begins to execute the scripts needed to bring the system to multiuser mode. When you boot the system into single-user mode, **init** runs the startup scripts after you terminate the single-user shell by typing **exit** or <Control-D>.

In multiuser mode, **init** is responsible for making sure that processes are available to handle logins on every login-enabled device. Logins on serial ports are generally handled by some variant of **getty** (e.g., **agetty**, **mgetty**, or **mingetty**; see page 857 for details). **init** also supervises a graphical login procedure that allows users to log directly in to X Windows.

In addition to its login management duties, **init** also has the responsibility to exorcise undead zombie processes that would otherwise accumulate on the system. **init**'s role in this process is described on page 56.

*See page 856 for more information about the **inittab** file.*

init defines several "run levels" that determine what set of system resources should be enabled. There are seven levels, numbered 0 to 6. The name "s" is recognized as a synonym for level 1 (single-user mode). The characteristics of each run level are defined in the **/etc/inittab** file.

init usually reads its initial run level from the **/etc/inittab** file, but the run level can also be passed in as an argument from the boot loader. If "s" is specified, **init** enters single-user mode. Otherwise, it scans **/etc/inittab** for entries that apply to the requested run level and executes their corresponding commands.

The **telinit** command changes **init**'s run level once the system is up. For example, **telinit 4** forces **init** to go to run level 4 (which is unused on our example systems). **telinit**'s most useful argument is **q**, which causes **init** to reread the **/etc/inittab** file.

Linux distributions generally implement an additional layer of abstraction on top of the basic run-level mechanism provided by **init**. The extra layer allows individual software packages to install their own startup scripts without modifying the system's generic **inittab** file. Bringing **init** to a new run level causes the appropriate scripts to be executed with the arguments **start** or **stop**.

A more complete discussion of **init** and startup scripts begins on page 33.

29.2 CRON AND ATD: SCHEDULE COMMANDS

The **cron** daemon (known as **crond** on Red Hat) is responsible for running commands at preset times. It accepts schedule files ("crontabs") from both users and administrators.

cron is frequently employed for administrative purposes, including management of log files and daily cleanup of the filesystem. In fact, **cron** is so important to system administrators that we have devoted an entire chapter to it. That chapter, *Periodic Processes*, begins on page 150.

The **atd** daemon runs commands scheduled with the **at** command. Most versions of Linux also include the **anacron** scheduler, which executes jobs at time intervals rather than at specific times. **anacron** is particularly useful on systems that are not always turned on, such as laptops.

29.3 XINETD AND INETD: MANAGE DAEMONS

xinetd and **inetd** are daemons that manage other daemons. They start up their client daemons when there is work for them to do and allow the clients to die gracefully once their tasks have been completed.

The traditional version of **inetd** comes to us from the UNIX world, but most Linux distributions have migrated to Panos Tsirigotis's **xinetd**, a souped-up alternative that incorporates security features similar to those formerly achieved through the use of **tcpd**, the "TCP wrappers" package. **xinetd** also provides better protection against denial of service attacks, better log management features, and a more flexible configuration language.

Unfortunately, **inetd**'s configuration file is not forward-compatible with that of **xinetd**. We first discuss the more common **xinetd** and then take a look at **inetd** in a separate section.

Among our example distributions, only Debian and Ubuntu use the standard **inetd**; RHEL, Fedora, and SUSE all default to **xinetd**. You can convert any system to use the nondefault daemon manager, but there's no compelling reason to do so.

xinetd and **inetd** only work with daemons that provide services over the network. To find out when someone is trying to access one of their clients, **xinetd** and **inetd** attach themselves to the network ports that would normally be managed by the quiescent daemons. When a connection occurs, **xinetd/inetd** starts up the appropriate daemon and connects its standard I/O channels to the network port. Daemons must be written with this convention in mind if they are to be compatible.

Some daemons (such as those associated with NIS and NFS) rely on a further layer of indirection known as the Remote Procedure Call (RPC) system. RPC was originally designed and implemented by Sun as a way of promoting the sharing of information in a heterogeneous networked environment. Port assignments for daemons that use RPC are managed by the **portmap** daemon, which is discussed later in this chapter.

Some daemons can be run in either the traditional fashion (in which they are started once and continue to run until the system shuts down) or through **xinetd/inetd**. Daemons discussed in this chapter are marked with an ⓘ if they are **xinetd/inetd**-compatible.

Because **xinetd/inetd** is responsible for managing many common network-based services, it plays an important role in securing your system. It's important to verify that only services you need and trust have been enabled. On a new system, you will almost certainly need to modify your default configuration to disable services that are unnecessary or undesirable in your environment.

Configuring xinetd

xinetd's main configuration file is traditionally **/etc/xinetd.conf**, although distributions commonly supply an **/etc/xinetd.d** configuration directory as well. Individual packages can drop their config files into this directory without worrying about overriding the configurations of other packages.

The example below shows the setting of default parameters and the configuration of an FTP service on a Red Hat Enterprise system.

```
defaults
{
    instances          = 60
    log_type           = SYSLOG authpriv
    log_on_success     = HOST PID
    log_on_failure     = HOST
    cps                = 25 30
}

service ftp
{
    # Unlimited instances because wu.ftpd does its own load management
    socket_type        = stream
    protocol           = tcp
    wait               = no
```

```
        user              = root
        server            = /usr/sbin/wu.ftpd
        server_args       = -a
        instances         = UNLIMITED
        only_from         = 128.138.0.0/16
        log_on_success    += DURATION
}

includedir /etc/xinetd.d
...
```

Table 29.1 provides a mini-glossary of parameters.

Table 29.1 xinetd configuration parameters (not an exhaustive list)

Parameter	Value	Meaning
bind	*ipaddr/host*	Interface on which to make this service available
cps	*num waittime*	Limits overall connections per second
disable	yes/no	Disables service; easier than commenting it out
include	*path*	Reads listed path as a supplemental config file
includedir	*path*	Reads all files in the specified directory
instances	*num* or UNLIMITED	Maximum number of simultaneous instances of a given service
log_on_failure	*special*[a]	Information to log for failures or access denials[b]
log_on_success	*special*[a]	Information to log for successful connections[b]
log_type	*special*[a]	Configures log file or syslog parameters
max_load	*num*	Disables service if load average > threshold
nice	*num*	Nice value of spawned server processes
no_access	*matchlist*	Denies service to specified IP addresses
only_from	*matchlist*	Accepts requests only from specified addresses
per_source	*num*	Limits number of instances per remote peer
protocol	tcp/udp	Service protocol
server	*path*	Path to server binary
server_args	*string*	Command-line arguments for server[c]
socket_type	stream/dgram	Uses stream for TCP services, dgram for UDP
user	*username*	User (UID) as whom the service should run
wait	yes/no	Should **xinetd** butt out until the daemon quits?

a. One or more values from a defined list too long to be worth reproducing in this table.

b. Note that the USERID directive used with these parameters causes **xinetd** to perform IDENT queries on connections, often resulting in significant delays.

c. Unlike **inetd**, **xinetd** does not require the server command to be the first argument.

Some **xinetd** parameters can accept assignments of the form += or -= (as seen in the log_on_success value for the FTP server) to modify the default values rather than replacing them outright. Only a few parameters are really required for each service.

Address match lists for the only_from and no_access parameters can be specified in several formats. Most useful are CIDR-format IP addresses with an explicit mask (as shown in the example) and host or domain names such as boulder.colorado.edu and .colorado.edu—note the preceding dot. Multiple specifications can be separated with a space (as in all **xinetd** lists).

xinetd can either log directly to a file or submit log entries to syslog. Since the volume of log information can potentially be quite high on a busy server, it may make sense to use direct-to-file logging for performance reasons. Keep in mind that logging to a file is less secure than logging to a remote server through syslog because a hacker that gains access to the local system can doctor the log files.

xinetd can provide some interesting services such as forwarding of requests to an internal host that is not visible to the outside world. It's worth reviewing **xinetd**'s man page to get an idea of its capabilities.

Configuring inetd

Debian and Ubuntu are the only major Linux distributions that still use the traditional **inetd**. This version of **inetd** consults **/etc/inetd.conf** to determine on which network ports it should listen. The config file includes much the same information as **xinetd.conf**, but it uses a tabular (rather than attribute/value list) format. Here's a (pared-down) example from a Debian system:

```
# Example /etc/inetd.conf - from a Debian system

#:INTERNAL: Internal services
#echo       stream  tcp      nowait     root     internal
#echo       dgram   udp      wait       root     internal
...
#time       stream  tcp      nowait     root     internal
#time       dgram   udp      wait       root     internal

#:STANDARD: These are standard services.
#:BSD: Shell, login, exec and talk are BSD protocols.

#:MAIL: Mail, news and uucp services.
imap2       stream  tcp      nowait     root     /usr/sbin/tcpd /usr/sbin/imapd
imaps       stream  tcp      nowait     root     /usr/sbin/tcpd /usr/sbin/imapd

#:INFO: Info services
ident       stream  tcp      wait       identd   /usr/sbin/identd identd
...
#:OTHER: Other services
swat        stream  tcp      nowait.400 root     /usr/sbin/swat swat
# finger    stream  tcp      nowait     nobody   /usr/sbin/tcpd in.fingerd -w
391002/1-2  stream  rpc/tcp  wait       root     /usr/sbin/famd fam
...
```

The first column contains the service name. **inetd** maps service names to port numbers by consulting either the **/etc/services** file (for TCP and UDP services) or the **/etc/rpc** file and **portmap** daemon (for RPC services). RPC services are identified by names of the form *name/num* and the designation rpc in column three. In the example above, the last line is an RPC service.

The only other RPC service that is commonly managed by **inetd** is **mountd**, the NFS mount daemon. Linux distributions seem to run this daemon the old-fashioned way (by starting it at boot time), so you may have no RPC services at all in your **inetd.conf** file.

On a host with more than one network interface, you can preface the service name with a list of comma-separated IP addresses or symbolic hostnames to specify the interfaces on which **inetd** should listen for service requests. For example, the line

```
inura:time      stream    tcp    nowait   root     internal
```

provides the time service only on the interface associated with the name inura in DNS, NIS, or the **/etc/hosts** file.

The second column determines the type of socket that the service will use and is invariably stream or dgram. stream is used with TCP (connection-oriented) services, and dgram is used with UDP; however, some services use both, e.g., **bind**.

The third column identifies the communication protocol used by the service. The allowable types are listed in the **protocols** file (usually in **/etc**). The protocol is almost always tcp or udp. RPC services prepend rpc/ to the protocol type, as with rpc/tcp in the preceding example.

If the service being described can process multiple requests at one time (rather than processing one request and exiting), column four should be set to wait. This option allows the spawned daemon to take over management of the port as long as it is running; **inetd** waits for the daemon to exit before resuming its monitoring of the port. The opposite of wait is nowait; it makes **inetd** monitor continuously and fork a new copy of the daemon each time it receives a request. The selection of wait or nowait must correspond to the daemon's actual behavior and should not be set arbitrarily. When configuring a new daemon, check the **inetd.conf** file for an example configuration line or consult the man page for the daemon in question.

The form nowait.400, used in the configuration line for **swat**, indicates that **inetd** should spawn at most 400 instances of the server daemon per minute. The default is more conservative, 40 instances per minute. Given the nature of this service (a web administration tool for Samba), it's not clear why the throttle threshold was raised.

The fifth column gives the username under which the daemon should run. It's always more secure to run a daemon as a user other than root if that is possible. In the example above, **in.fingerd** would run as the user nobody (if the line were not commented out).

The remaining fields give the fully qualified pathname of the daemon and its command-line arguments. The keyword internal indicates services whose implementations are provided by **inetd** itself.

Many of the service entries in this example run their daemons by way of **tcpd** rather than executing them directly. **tcpd** logs connection attempts and implements access control according to the source of the connection attempt. In general, all services should be protected with **tcpd**. This example configuration presents a potential security problem because **swat**, a file sharing configuration utility, is not protected.[2]

In the default **inetd.conf** shipped with Debian, the servers for **rlogin**, **telnet**, **finger**, and **rexec** are no longer even listed. See the section *Miscellaneous security issues* on page 685 for more security-related information.

See Chapter 10 for more information about syslog.

After you edit **/etc/inetd.conf**, send **inetd** a HUP signal to tell it to reread its configuration file and implement any changes you made. After signalling, wait a moment and then check the log files for error messages related to your changes (**inetd** logs errors to syslog under the "daemon" facility). Test any new services you have added to be sure they work correctly.

The services file

After adding a new service to **inetd.conf** or **xinetd.conf**, you may also need to make an entry for it in the **/etc/services** file. This file is used by several standard library routines that map between service names and port numbers. **xinetd** actually allows you to specify the port number directly, but it's always a good idea to maintain a master list of ports in the **services** file.

For example, when you type the command

```
$ telnet anchor smtp
```

telnet looks up the port number for the smtp service in the **services** file. Most systems ship with all the common services already configured; you need only edit the **services** file if you add something new.

The **services** file is used only for bona fide TCP/IP services; similar information for RPC services is stored in **/etc/rpc**.

Here are some selected lines from a **services** file (the original is ~570 lines long):

```
tcpmux    1/tcp                 # TCP port multiplexer
echo      7/tcp
echo      7/udp
…
ssh       22/tcp                #SSH Remote Login Protocol
ssh       22/udp                #SSH Remote Login Protocol
smtp      25/tcp    mail
rlp       39/udp    resource    # resource location
name      42/tcp                # IEN 116
domain    53/tcp                # name-domain server
domain    53/udp
…
```

2. If you are not using **tcpd** to protect a service, the daemon's first command-line argument should always be the short name of the daemon itself. This requirement is not a peculiarity of **inetd** but a traditional UNIX convention that is normally hidden by the shell.

The format of a line is

```
name        port/proto  aliases        # comment
```

Services are generally listed in numerical order, although this order is not required. *name* is the symbolic name of the service (the name you use in the **inetd.conf** or **xinetd.conf** file). The *port* is the port number at which the service normally listens; if the service is managed by **inetd**, it is the port that **inetd** will listen on.[3]

The *proto* stipulates the protocol used by the service; in practice, it is always tcp or udp. If a service can use either UDP or TCP, a line for each must be included (as with the ssh service above). The *alias* field contains additional names for the service (e.g., whois can also be looked up as nicname).

portmap: map RPC services to TCP and UDP ports

portmap maps RPC service numbers to the TCP/IP ports on which their servers are listening. When an RPC server starts up, it registers itself with **portmap**, listing the services it supports and the port at which it can be contacted. Clients query **portmap** to find out how to get in touch with an appropriate server.

This system allows a port to be mapped to a symbolic service name. It's basically another level of abstraction above the **services** file, albeit one that introduces additional complexity (and security issues) without solving any real-world problems.

If the **portmap** daemon dies, all the services that rely on it (including **inetd** and NFS) must be restarted. In practical terms, this means that it's time to reboot the system. **portmap** must be started before **inetd** for **inetd** to handle RPC services correctly.

29.4 Kernel daemons

For architectural reasons, a few parts of the Linux kernel are managed as if they were user processes. On older kernels, these processes could be identified by their low PIDs and names that start with **k**, such as **kupdate**, **kswapd**, **keventd**, and **kapm**. The naming is less consistent under the 2.6 kernels, but **ps** always shows the names of kernel threads in square brackets.

For the most part, these processes deal with various aspects of I/O, memory management, and synchronization of the disk cache. They cannot be manipulated by the system administrator and should be left alone.[4]

Table 29.2 on the next page briefly summarizes the functions of the major daemons in the current complement. Daemons that include an N parameter in their names

3. Port numbers are not arbitrary. All machines must agree about which services go with which ports; otherwise, requests will constantly be directed to the wrong port. If you are creating a site-specific service, pick a high port number (greater than 1023) that is not already listed in the **services** file.
4. If you are familiar with the implementation of the kernel, it is occasionally useful to change these processes' execution priorities. However, this is not a standard administrative task.

(as shown by **ps**) run separately on each CPU of a multi-CPU system; the N tells you which copy goes with which CPU.

Table 29.2 Major kernel daemons (2.6 kernels)

Daemon	Function
ksoftirqd/*N*	Handles software interrupts when the load is high
kacpid	Deals with the ACPI subsystem
kblockd/*N*	Blocks subsystem work
aio/*N*	Retries asynchronous I/Os
kswapd*N*	Moves pages to swap
ata/*N*	Does processing for serial ATA support
scsi_eh_*N*	Performs SCSI error handling
kjournald	Supports journaling filesystems
events/*N*	Does generic work queue processing

Another system daemon in this category, albeit one with a nonstandard name, is **mdrecoveryd**. It's part of the "multiple devices," implementation, more commonly known as RAID.

klogd: read kernel messages

klogd is responsible for reading log entries from the kernel's message buffer and forwarding them to syslog so that they can be routed to their final destination. It can also process messages itself if configured to do so. See *Kernel and boot-time logging* on page 206 for more information.

29.5 Printing daemons

Several printing systems are in common use, and each has its own family of commands and daemons that provide printing-related services. In some cases the families have been hybridized; in others cases, multiple variants run on a single system.

cupsd: scheduler for the Common UNIX Printing System

See Chapter 23 for more information about CUPS.

CUPS provides a portable printing facility by implementing version 1.1 of the Internet Printing Protocol. It allows remote users to print to their offices (or vice versa) by using a web interface. CUPS has become quite popular and is most systems' default printing manager. It is flexible enough to allow for remote authentication.

lpd: manage printing

lpd is responsible for the old-style BSD print spooling system. It accepts jobs from users and forks processes that perform the actual printing. **lpd** is also responsible for transferring print jobs to and from remote systems. **lpd** can sometimes hang and then must be manually restarted.

Your system might have either the original flavor of **lpd** or the extra-crispy version that's part of the LPRng package. See Chapter 23, *Printing*, for more information about these alternatives.

29.6 File service daemons

The following daemons are part of the NFS or Samba file sharing systems. We give only a brief description of their functions here. NFS is described in detail in Chapter 16, and Samba is covered starting on page 828.

rpc.nfsd: serve files

rpc.nfsd runs on file servers and handles requests from NFS clients. In most NFS implementations, **nfsd** is really just a part of the kernel that has been dressed up as a process for scheduling reasons. Linux actually sports two different implementations, one of which follows this convention and one of which runs in user space. The kernel implementation is more popular and is most distributions' default.

rpc.nfsd accepts a single argument that specifies how many copies of itself to fork. Some voodoo is involved in picking the correct number of copies; see page 492.

rpc.mountd: respond to mount requests

rpc.mountd accepts filesystem mount requests from potential NFS clients. It verifies that each client has permission to mount the requested directories. **rpc.mountd** consults the **/var/state/nfs/xtab** file to determine which applicants are legitimate.

amd and automount: mount filesystems on demand

amd and **automount** are NFS automounters, daemons that wait until a process attempts to use a filesystem before they actually mount it. The automounters later unmount the filesystems if they have not been accessed in a specified period of time.

The use of automounters is very helpful in large environments where dozens or hundreds of filesystems are shared on the network. Automounters increase the stability of the network and reduce configuration complexity since all systems on the network can share the same **amd** or **automountd** configuration. We cover the use of the standard Linux automounter in detail starting on page 497.

rpc.lockd and rpc.statd: manage NFS locks

Although **rpc.lockd** and **rpc.statd** are distinct daemons, they always run as a team. **rpc.lockd** maintains advisory locks (a la **flock** and **lockf**) on NFS files. **rpc.statd** allows processes to monitor the status of other machines that are running NFS. **rpc.lockd** uses **rpc.statd** to decide when to attempt to communicate with a remote machine.

rpciod: cache NFS blocks

rpciod caches read and write requests on NFS clients. It performs both read-ahead and write-behind buffering and greatly improves the performance of NFS. This daemon is analogous to the **biod** and **nfsiod** daemons found on other systems, although it is structurally somewhat different.

rpc.rquotad: serve remote quotas

rpc.rquotad lets remote users check their quotas on filesystems they have mounted with NFS. The actual implementation of quota restrictions is still performed on the server; **rpc.rquotad** just makes the **quota** command work correctly.

smbd: provide file and printing service to Windows clients

smbd is the file and printer server in the Samba suite. It provides file and printer sharing service through the Windows protocol known variously as SMB or CIFS. See page 828 for more details.

nmbd: NetBIOS name server

nmbd is another component of Samba. It replies to NetBIOS name service requests generated by Windows machines. It also implements the browsing protocol that Windows machines use to populate the My Network Places folder and makes disks shared from the local host visible there. **nmbd** can also be used as a WINS server.

29.7 Administrative database daemons

Several daemons are associated with Sun's NIS administrative database system, which is described in Chapter 17, *Sharing System Files*. Although NIS originated at Sun, it is now used on many other vendors' systems as well, including Linux.

ypbind: locate NIS servers

The **ypbind** daemon runs on all NIS clients and servers. It locates an NIS server to which queries can be directed. **ypbind** does not actually process requests itself; it just tells client programs which server to use.

ypserv: NIS server

ypserv runs on all NIS servers. **ypserv** accepts queries from clients and responds with the requested information. See page 517 for information on how to configure the machines that run **ypserv**.

rpc.ypxfrd: transfer NIS databases

rpc.ypxfrd efficiently transfers NIS databases to slave servers. A slave initiates a transfer with the **ypxfr** command. Whenever a database is changed on the master, it should immediately be pushed out to all the slaves so that the NIS servers remain consistent with one another.

lwresd: lightweight resolver library server

lwresd provides a quick method of caching address-to-hostname and hostname-to-address lookups. It's contacted by a stub resolver that is part of the system's standard libraries and is called directly by many programs. The library and daemon communicate through a simple UDP protocol.

nscd: name service cache daemon

nscd caches the results of calls to the standard C library routines in the **getpw***, **getgr***, and **gethost*** families, which look up data that was traditionally stored in the **passwd**, **group**, and **hosts** files. These days, the range of potential sources is larger and includes options such as NIS and DNS. **nscd** does not actually know where the data comes from; it simply caches results and uses them to short-circuit subsequent library calls. Caching policy is set in the **/etc/nscd.conf** file.

29.8 ELECTRONIC MAIL DAEMONS

In addition to the core **sendmail** and Postfix mail delivery systems, which are both in widespread use, several daemons facilitate remote access to mailboxes.

sendmail: transport electronic mail

sendmail's tasks include accepting messages from users and remote sites, rewriting addresses, expanding aliases, and transferring mail across the Internet. **sendmail** is an important and very complex daemon. Refer to Chapter 18, *Electronic Mail*, for the complete scoop.

smtpd: Simple Mail Transport Protocol daemon

smtpd listens on port 25 for incoming email messages and forwards them to your back-end transport system for further processing. See pages 540 and 624 for more information about the use of **smtpd** in the **sendmail** and Postfix systems.

popd: basic mailbox server

The **popd** daemon implements the Post Office Protocol (POP). This protocol is commonly used by non-Linux systems to receive electronic mail.

imapd: deluxe mailbox server

The **imapd** daemon implements the Internet Message Access Protocol, IMAP, which is a more festive and featureful alternative to POP. It allows PC-based users (or Linux users with IMAP-enabled mail readers) to access their email from a variety of locations, with mail folders being stored on the Linux server. Check out www.imap.org for more information about IMAP.

29.9 Remote login and command execution daemons

The ability to log in and execute commands over the net was one of the earliest motivations for the development of UNIX networking, and this facility is still a bread-and-butter component of system administration today. Unfortunately, it took the UNIX community several decades to achieve a mature appreciation of the security implications of this technology. Modern production systems should be using SSH (**sshd**) and virtually nothing else.

sshd: secure remote login server

sshd provides services that are similar to **in.rlogind**, but its sessions are transported (and authenticated) across an encrypted pipeline. A variety of encryption algorithms are available. Because of the harsh environment of the Internet today, you must allow shell access from the Internet *only* through a daemon such as this—*not* **in.rlogind** or **in.telnetd**. You can find more information about **sshd** starting on page 697.

in.rlogind: obsolete remote login server

in.rlogind was the long-ago standard for handling remote logins. When invoked by **inetd**, it tries to automatically authenticate the remote user by examining the local user's **~/.rhosts** file and the system-wide **/etc/hosts.equiv**. If automatic authentication is successful, the user is logged in directly. Otherwise, **in.rlogind** executes the **login** program to prompt the user for a password. Because of its cheap 'n' easy authentication, **in.rlogind** is a major security hazard. See page 685 for more comments on this subject.

in.telnetd: yet another remote login server

in.telnetd is similar to **in.rlogind**, except that it uses the TELNET protocol. This protocol allows the two sides (client and server) to negotiate flow control and duplex settings, making it a better choice than **in.rlogind** for links that are slow or unreliable. Like **rlogin**, **telnet** transmits plaintext passwords across the network. Its use is therefore discouraged in modern networks. However, many non-Linux systems support **telnet**.

in.rshd: remote command execution server

in.rshd handles remote command execution requests from **rsh** and **rcmd**. The authentication process enforced by **in.rshd** is similar to that of **in.rlogind**, except that if automatic authentication does not work, **in.rshd** denies the request without allowing the user to supply a password. **in.rshd** is also the server for **rcp** (remote copy). Like **in.rlogind**, **in.rshd** has become something of a security albatross and is invariably disabled. See page 685 for more information.

29.10 Booting and configuration daemons

In the 1980s, the UNIX world was swept by a wave of diskless workstation mania. These machines booted entirely over the network and performed all their disk I/O through a remote filesystem technology such as NFS. As disk prices dropped and speeds increased, interest in diskless workstations quickly faded. They could come back into fashion at any moment, however, like the platform shoes of the 1970s. The two main remnants of the diskless era are a plethora of daemons designed to support diskless systems and the bizarre organization of most vendors' filesystems.

For the curious, we discuss diskless systems themselves in some additional detail starting on page 232.

Although diskless workstations are not common anymore, their booting protocols have been usurped by other devices. Most manageable network hubs and network printers boot by using some combination of the services listed in this section.

dhcpd: dynamic address assignment

The Dynamic Host Configuration Protocol (DHCP) provides PCs, laptops, and other "mobile" platforms with information about their IP addresses, default gateways, and name servers at boot time. **dhcpd** is the daemon that implements this service under Linux. You can find more information about DHCP on page 311. A fancier elaboration of DHCP called PXE (Pre-boot eXecution Environment) helps compatible machines boot from the network without the need for a local boot device; see page 224 for more details.

in.tftpd: trivial file transfer server

in.tftpd implements a file transfer protocol similar to that of **ftpd**, but much, much simpler. Many diskless systems use TFTP to download their kernels from a server. **in.tftpd** does not perform authentication, but it is normally restricted to serving the files in a single directory (usually **/tftpboot**). Since anything placed in the TFTP directory is accessible to the entire network, the directory should contain only boot files and should not be publicly writable.

rpc.bootparamd: advanced diskless life support

rpc.bootparamd uses the **/etc/bootparams** file to tell diskless clients where to find their filesystems. **rpc.bootparamd** service is often used by machines that get their IP addresses by using RARP and that use NFS to mount their filesystems.

hald: hardware abstraction layer (HAL) daemon

hald collects information about the system's hardware from several sources. It provides a live device list through D-BUS.

udevd: serialize device connection notices

udevd is a minor part of the **udev** dynamic device-naming system. It allows for the proper serialization of hot-plug events, which the kernel can sometimes communicate out of order to user space.

29.11 Other network daemons

The following daemons all use Internet protocols to handle requests. However, many of these “Internet” daemons actually spend the majority of their time servicing local requests.

talkd: network chat service

Connection requests from the **talk** program are handled by **talkd**. When it receives a request, **talkd** negotiates with the other machine to set up a network connection between the two users who have executed **talk**.

snmpd: provide remote network management service

snmpd responds to requests that use the Simple Network Management Protocol (SNMP) protocol. SNMP standardizes some common network management operations. See page 659 for more information about SNMP.

ftpd: file transfer server

See page 734 for more information about ***ftpd****.*

ftpd is the daemon that handles requests from **ftp**, the Internet file transfer program. Many sites disable it, usually because they are worried about security. **ftpd** can be set up to allow anyone to transfer files to and from your machine.

A variety of **ftpd** implementations are available for Linux systems. If you plan to run a high-traffic server or need advanced features such as load management, it might be wise to investigate the alternatives to your distribution’s default **ftpd**.

WU-FTPD, developed at Washington University, is one of the most popular alternatives to the standard **ftpd**. See www.wu-ftpd.org for more information.

rsyncd: synchronize files among multiple hosts

rsyncd is really just a link to the **rsync** command; the **--daemon** option turns it into a server process. **rsyncd** facilitates the synchronization of files among hosts. It’s essentially an efficient and security-aware version of **rcp**. **rsync** is a real treasure trove for system administrators, and in this book we’ve described its use in a couple of different contexts. See page 508 for general information and some tips on using **rsync** to share system files. **rsync** is also a large part of many sites’ internal installation processes.

routed: maintain routing tables

routed maintains the routing information used by TCP/IP to send and forward packets on a network. **routed** deals only with dynamic routing; routes that are statically defined (that is, wired into the system’s routing table with the **route** command) are never modified by **routed**. **routed** is relatively stupid and inefficient, and we recommend its use in only a few specific situations. See page 343 for a more detailed discussion of **routed**.

gated: maintain complicated routing tables

gated understands several routing protocols, including RIP, the protocol used by **routed**. **gated** translates routing information among various protocols and is very configurable. It can also be much kinder to your network than **routed**. See page 344 for more information about **gated**.

named: DNS server

named is the most popular server for the Domain Name System. It maps hostnames into network addresses and performs many other feats and tricks, all using a distributed database maintained by **named**s everywhere. Chapter 15, *DNS: The Domain Name System*, describes the care and feeding of **named**.

syslogd: process log messages

See page 209 for more information about syslog.

syslogd acts as a clearing house for status information and error messages produced by system software and daemons. Before **syslogd** was written, daemons either wrote their error messages directly to the system console or maintained their own private log files. Now they use the **syslog** library routine to transfer the messages to **syslogd**, which sorts them according to rules established by the system administrator.

in.fingerd: look up users

in.fingerd provides information about the users that are logged in to the system. If asked, it can also provide a bit more detail about individual users. **in.fingerd** does not really do much work itself: it simply accepts lines of input and passes them to the local **finger** program.

finger can return quite a bit of information about a user, including the user's login status, the contents of the user's GECOS field in **/etc/passwd**, and the contents of the user's **~/.plan** and **~/.project** files.

If you are connected to the Internet and are running **in.fingerd**, anyone in the world can obtain this information. **in.fingerd** has enabled some really neat services (such as the Internet white pages), but it has also enabled people to run a variety of scams, such as finding people to cold-call and prospecting for spammable addresses. Some sites have responded to this invasion by turning off **in.fingerd**, while others just restrict the amount of information it returns. Don't assume that because **in.fingerd** is simple, it is necessarily secure—a buffer overflow attack against this daemon was exploited by the original Internet worm of 1988.

httpd: World Wide Web server

httpd lets your site become a web server. **httpd** can send text, pictures, and sound to its clients. See Chapter 21, *Web Hosting and Internet Servers*, for more information about serving up web pages.

29.12 NTPD: TIME SYNCHRONIZATION DAEMON

As computers have grown increasingly interdependent, it has become more and more important for them to share a consistent idea of time. Synchronized clocks are essential for correlating log file entries in the event of a security breach, and they're also important for a variety of end-user applications, from joint development of software projects to the processing of financial transactions.

ntpd[5] implements the Network Time Protocol, which allows computers to synchronize their clocks to within milliseconds of each other. The first NTP implementation started around 1980 with an accuracy of only several hundred milliseconds. Today, a new kernel clock model can keep time with a precision of up to one nanosecond. The latest version of the protocol (version 4, documented in RFC2783) maintains compatibility with the previous versions and adds easy configuration and some security features.

NTP servers are arranged in a hierarchy, each level of which is called a "stratum." The time on stratum 1 servers is typically slaved to an external reference clock such as a radio receiver or atomic clock. Stratum 2 servers set their clocks from Stratum 1 servers and act as time distribution centers. Up to 16 strata are provided for. To determine its own stratum, a time server simply adds 1 to the stratum of the highest-numbered server to which it synchronizes. A 1999 survey of the NTP network by Nelson Minar indicated that there were (at that time) 300 servers in stratum 1; 20,000 servers in stratum 2; and more than 80,000 servers in stratum 3.[6]

Today, NTP clients can access a number of reference time standards, such as those provided by WWV and GPS. A list of authoritative U.S. Internet time servers maintained by the National Institute of Standards and Technology can be found at

www.boulder.nist.gov/timefreq/service/time-servers.html

Most ISPs maintain their own set of time servers, which should be closer in network terms for their downstream clients (and if NTP works correctly, just as accurate).

ntpd implements both the client and server sides of the NTP protocol. It reads **/etc/ntp.conf** at startup. In the config file you can specify access, client networks, time servers, multicast clients, general configuration, and authentication; but don't be scared off—it's all pretty self-explanatory.

Debian and Ubuntu don't seem to include **ntpd** by default, but it's readily available through **apt-get**. You can also obtain the current software from ntp.isc.org.

You can also use the quick and dirty **ntpdate** utility to set the system's clock from an NTP server. This is a less desirable solution than **ntpd** because it can make the flow of time appear discontinuous. It is especially harmful to set the clock back suddenly, since programs sometimes assume that time is a monotonically increasing function.

5. This daemon was also known as **xntpd** in earlier incarnations.
6. See www.media.mit.edu/~nelson/research/ntp-survey99

ntpd uses the gentler **adjtimex** system call to smooth the adjustment of the system's clock and prevent large jumps backward or forward. **adjtimex** biases the speed of the system's clock so that it gradually falls into correct alignment. When the system time matches the current objective time, the bias is cancelled and the clock runs normally.

29.13 Exercises

☆ **E29.1** Using **ps**, determine which daemons are running on your system. Also determine which daemons are available to run through **inetd.conf** or **xinetd.conf**. Combine the lists and describe what each daemon does, where it is started, whether multiple copies can (or do) run at the same time, and any other attributes you can glean.

☆ **E29.2** In the lab, install and set up the network time daemon, **ntpd**.

a) How do you tell if your system has the correct time?

b) Using the **date** command, manually set your system time to be 15 seconds slow. How long does it (or will it) take for the time become correct?

c) Manually set your system time a month ahead. How does **ntpd** respond to this situation?

(Requires root access.)

☆☆ **E29.3** In the lab, use a tool such as **netstat** to determine what ports are in a "listening" state on your machine.

a) How can you reconcile the **netstat** information with what is found in **inetd.conf** or **xinetd.conf**? If there is a discrepancy, what is going on?

b) Install the **nmap** tool on a different machine. Run a port scan targeting your system to verify what you learned in part a. What (if any) additional information did you learn from **nmap** that wasn't obvious from **netstat**? (See page 688 for more information about **nmap**.)