In this chapter:

- *What's a Dedicated Server?*
- *Account Provisioning*
- *Mission Restriction*
- *The Ultimate in Dedicated Servers*

14

Running a Dedicated Server

In this chapter, we'll discuss some of the motivations and challenges associated with running a dedicated IMAP server.

What's a Dedicated Server?

One of Unix's greatest strengths can also turn into a weakness. Just because you can provide a multitude of dissimilar services on a single server doesn't mean you should. Reduce the total number of services you offer on one server to one or two, and you may increase manageability, robustness, and security several-fold. On your mail server, this could directly translate to happier end users.

The goal of a dedicated server is to minimize administration and maintenance overhead while maximizing the performance of the service to which the server is dedicated. Then what is a dedicated server? It's a host tuned to provide a single service. A dedicated IMAP server, for example, would provide only IMAP services. It would not provide shell accounts (other than accounts required for system maintenance), IRC, Usenet, or any other service that is not directly required to provide IMAP service. Simply put, a dedicated IMAP server receives mail, deposits it in the mailstore, and provides access to the mailstore exclusively by way of IMAP.

Account Provisioning

Without shell accounts on a dedicated server, how does a user perform routine tasks such as changing her password or setting up mail forwarding? Dedicated servers, by definition, have no non-administrative shell accounts. Once you've done away with shell accounts, you're presented with the challenge of finding a

way to provide the shell services by other means. Your solution should be both user-friendly and ubiquitous, and it should use your available resources responsibly.

If you're confident that all your users are on the same platform, you could employ various platform-specific provisioning solutions, such as an X-based or Windows application, Microsoft Exchange form, or your own home-brew application. Trust us, though—long-term maintenance costs of those solutions far outweigh the immediate gratification you'll receive.

A Web Solution

A good way to handle provisioning on a dedicated server is to bring up a provisioning web site.

There is a web of distractions out there, ranging from reliance on browser-specific features to various early attempts at standardized client-side scripting. Administrators should remain vigilant against developing a provisioning site that becomes arcane and proprietary. Stick to your guns and develop a straightforward, simple site that permits the user to perform simple actions (e.g., a password change or quota check). Even with such a simple web site, you'll find that you've eliminated nearly all of the need for users to have shell accounts. The remainder of your reasons for having user shell accounts, assuming those reasons are not IMAP related, could subsequently be addressed by bringing up a modest shell account host, such as a commodity PC running Linux.

There are five issues at the core of any provisioning web site: security, authentication, ease of use, system load, and permanence.

Security

You are your own best judge of what security issues are relevant to your particular provisioning site. Common critical issues include security of the data stream between the browser and server, security of the data on the provisioning system itself, and the security of the implementation.

Right now, the most practical way to secure your data stream is by using SSL (Secure Sockets Layer). If you require all users to use an SSL-enabled browser with encryption using 128-bit or larger keys, you've secured your data stream sufficiently. The Herculean effort required to compromise your data stream outweighs the value of the information reward to be gained. By using SSL you have the added benefit of encrypting password strings sent from the browser to the server. If you opt not to use SSL, your users should be appropriately warned that their passwords will be transmitted over the network in cleartext.

With all the secure plumbing in place, it's equally important that the provisioning data on your server be secure as well. One well-known national ISP gained notoriety a few years ago when a hacker was able to retrieve a cleartext file containing hundreds of customer names and credit card numbers.

When developing HTML forms, something you'll want to be attentive to is using the POST instead of GET method in your forms. The GET method conveys form variables on the URL command line, making them easy to retrieve by paging through the browser's URL history. The POST method, on the other hand, conveys those variables in the input stream to the server. If you use the POST method, subsequent users of the customer's machine can't go through the browser history and collect information useful in gaining access to the customer's account.

Some sites may decide to use HTTP cookies to allow a user to log in to the provisioning system and perform tasks without having to authenticate for each task. If you employ HTTP cookies, the cookie should expire after a brief period. Although it's possible to do so, subsequent users of a workstation will find it more difficult to masquerade as a user using someone else's cookie if that cookie has expired. HTTP cookies have a secure flag that, when set, will send the cookie to the server only if the CGI request is occurring on an SSL channel. You'll probably want to set the cookie's secure flag, although it's not necessary if your provisioning system is exclusively available via SSL.

We advise that you try to have as many levels of security as practical. It's always a good thing, in the planning, to assume that one or two levels of your security will be compromised. Ask yourself "what if" questions. If you don't have any shell accounts on your standalone server, but someone manages to get shell access anyway, are the permissions on critical directories and files closed down far enough that someone with non-root access would find such access useless?

Authentication

A provisioning web site must be able to authenticate its users. There are numerous ways to authenticate users of web sites, such as HTTP cookies, HTML form variables, or truly distasteful methods like assuming a user always logs in from a particular IP address. Ultimately, the correct method of authentication for your site would be the one that provides you with the most security and is most consistent with your existing authentication environment. What we're trying to get at here is that users are most likely to embrace an interface that's easy for them to use. With regard to authentication, that frequently means avoiding multiple passwords per user whenever possible.

CGI Scripts for Common Tasks

As a starting point, here are some Perl CGI scripts that will do some of the tasks we've mentioned. The examples in this section require you to install a web server, Perl 5 (*http://www.perl.com/*), and the CGI.pm Perl module (*http://www.cpan.org/modules/by-module/CGI/CGI.pm-2.56.tar.gz*). CGI.pm is a Perl library used to make writing CGI scripts easier. You'll find documentation on CGI.pm at *http://stein.cshl.org/WWW/software/CGI/*.

It's highly recommended that the web server be SSL-enabled. Details on how to set up an SSL-enabled web server are beyond the scope of this book, but a quick, simple, and free way is to use Apache-SSL (*http://www.apache-ssl.org/*) and OpenSSL (*http://www.openssl.org/*).

Changing a password

The password change utility described in this section uses a freely available password-changing CGI program called *chpasswd*. *chpasswd* is available for download from the *chpasswd* author's site (*http://sic.popnet.pl/~mlody/chpasswd/chpasswd-1.3.tar.gz*) or from FreshMeat (*http://freshmeat.net/appindex/web/tools.html*).

There is a multitude of free web-based password-changing utilities available on the Net. *chpasswd* is mentioned here because it's a utility that's used to change standard Unix and shadow passwords, and thus it fills the needs of sites that rely on Unix authentication. Many of the utilities we found employ a *setuid* Perl or shell script to perform the password change. *chpasswd*, on the other hand, is a compiled executable. Although the executable is *setuid*, *setuid* executables don't pose as many risks as *suid* scripts because scripts depend on external programs that can be replaced, for example, with copies of *bash* to provide easy root access to malevolent users. *chpasswd* has the added security of consulting a *deny* file (*/etc/www.deny*) before processing any request. Users listed in the */etc/www.deny* (*root*, for example) are not allowed to change their password using the CGI. *chpasswd* logs the results of every password change request to syslog.

chpasswd was written for Linux, but we installed it and ran it successfully under Solaris with no problems. Since the *chpasswd* program uses the operating system's native *crypt* function, it should work equally well with other flavors of Unix. Keep in mind that *crypt* supports weak cryptography and thus provides only minimal security.

To build *chpasswd*, download and unpack the source distribution and run the *configure.sh* script. The script will ask you for the path to your *cgi-bin* directory and the HTTP path to the *chpasswd* CGI script. After running the *configure.sh* script, run *make* and *make install*. *make install* will copy the *chpasswd.cgi* program into your *cgi-bin* directory and will install in the source directory a

rudimentary password change HTML form that submits input to the CGI. Example 14-1 is a slightly modified version of that form.

Example 14-1. Change Password Form

```
<HTML>
<BODY>
<TITLE>Password Change</TITLE>

<H1>Change Your Password</H1>

<form method="POST" action="https://themullets.net/cgi-bin/chpasswd.cgi">

<PRE>
Username:             <input type="text" name="login">
Current password:     <input type="password" name="password">
New password:         <input type="password" name="newpassword">
Confirm new password: <input type="password" name="newpassword2">
<PRE>

<P>
<input type="submit" value="OK"> <input type="reset" value="RESET"><BR>
</FORM>
</BODY>
</HTML>
```

Checking disk quota

Sites that run the UW IMAP server often configure the server to store personal mail folders under a user's home directory. Because UW doesn't explicitly support the IMAP quota extension, UW sites usually fall back on operating-system disk quotas. In those circumstances, it's frequently handy to give the user a tool to check his quota. Example 14-2 and Example 14-3 are a CGI form and handler that allow the user to check his disk quota.

Example 14-2. Quota Check CGI Form

```
#!/usr/local/bin/perl

use CGI;

$query = new CGI;

print $query->header;
print $query->start_html(-title=>'Check Quota');
print $query->startform(-action=>"quota_results.cgi");

print $query->h1($query->center("Check Quota"));
print $query->hr;
print <<EOM;
<p>Enter your username and password and click the Check Quota button.<p>
EOM
```

Example 14-2. Quota Check CGI Form (continued)

```
print "Username: ", $query->textfield(-name=>'user'),          $query->br;
print "Password: ", $query->password_field(-name=>'password'), $query->p;
print $query->center($query->submit('action','Check Quota'));
print $query->hr;

print $query->endform;
print $query->end_html;
```

Example 14-3, the form handler, authenticates the user based on the username and password entered on the previous form, then calls an external program to get the user's quota information. We leave out the details of the password authentication for the sake of generality.

Example 14-3. Quota Check CGI Form Handler

```
#!/usr/local/bin/perl

use CGI qw(:standard);

$query = new CGI;
$user     = $query->param('user');
$password = $query->param('password');

print $query->header;
print $query->start_html(-title=>'Check Quota');
print $query->center($query->h1("Disk Quota Results"));
print $query->hr;

if (correct_pass("$user","$password") == 1) {

    $quota = `/opt/apache/cgi-bin/quota $user`;

    print <<EOF;
    Disk quota and usage for $user:<p>

    <pre>$quota</pre>
EOF
} else {
    print "Login incorrect. Go back and try again."
}

print $query->p,$query->hr;
print $query->end_html;
```

The program that does the actual quota check within the CGI is a compiled C *setuid* program that calls the Unix *quota* command. CGI scripts run as the owner of the web server process, typically the user *nobody*. *nobody* is an unprivileged user and, as such, cannot gather data on other users using Unix commands like *quota*. The *setuid* program, referred to as a wrapper, changes its process ownership to a

privileged user before executing the Unix command, then switches its ownership back to the original owner once the work is done.

There are other solutions to the challenges that arise from running your web server as an unprivileged user, some of them acceptable (carefully written setuid wrappers) and some truly dangerous (giving up and running the web server as *root*). Setuid programs have their own set of security problems, but the dangers are limited compared to other solutions. The GNU C Library documentation (*http://www.gnu.org/manual/glibc-2.0.6/html_mono/libc.html*) has an excellent set of guidelines for writing good setuid programs, with examples.

quota.c, the setuid wrapper source code, is shown in Example 14-4. It's important to note that the permissions on the compiled executable must have the setuid bit set, and the executable must be owned by *root*. If both conditions are not met, the program will not be able to change ownership of the process to the root and will not have sufficient permissions to run the *quota* command:

```
# gcc -o quota quota.c
# chown root:other quota
# chmod 4755 quota
# ls -l quota
-rwsr-xr-x   1 root     other        8660 Jan  2 20:54 quota*
```

Example 14-4. Quota Command Setuid Wrapper Program

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

static uid_t euid, ruid;

/* Restore the effective UID to its original value. */

void
do_setuid (void)
{
  int status;

  status = setreuid (ruid, euid);
  if (status < 0) {
    fprintf (stderr, "Couldn't set uid.\n");
    exit (status);
    }
}

/* Set the effective UID to the real UID. */

void
undo_setuid (void)
{
```

Example 14-4. Quota Command Setuid Wrapper Program (continued)

```
  int status;

  status = setreuid (euid, ruid);
  if (status < 0) {
    fprintf (stderr, "Couldn't set uid.\n");
    exit (status);
    }
}

/* Main program. */

int
main(int argc, char **argv)
{
    FILE *fp;
    int pid, pipefds[2];
    char *user = argv[1];
    static uid_t euid, ruid;

    if (argc != 2) { printf("Usage: %s user\n", *argv); exit(1); }

    /* Save the real and effective user IDs.  */
    ruid = getuid ();  euid = geteuid ();

    if (pipe(pipefds) < 0) {
        perror("pipe"); exit(1);
        }

    if ((pid = fork()) < 0) {
        perror("fork"); exit(1);
        }

    if (pid == 0) {
        close(0);
        dup(pipefds[0]);
        close(pipefds[0]);
        close(pipefds[1]);

        /* Set user to real userid (file owner) */

        do_setuid();
        execl("/usr/sbin/quota", "quota", "-v", user, (char *) 0);

        perror("exec");
        exit(1);
    }

    close (pipefds[0]);
    exit(0);
}
```

Figure 14-1 and Figure 14-2 are screen shots of the CGI form and handler results.

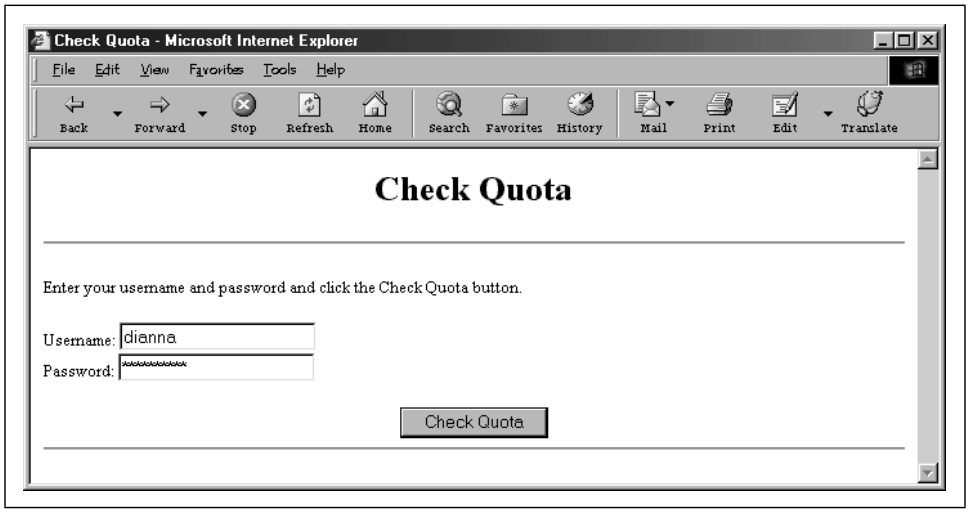

Figure 14-1. Quota CGI form

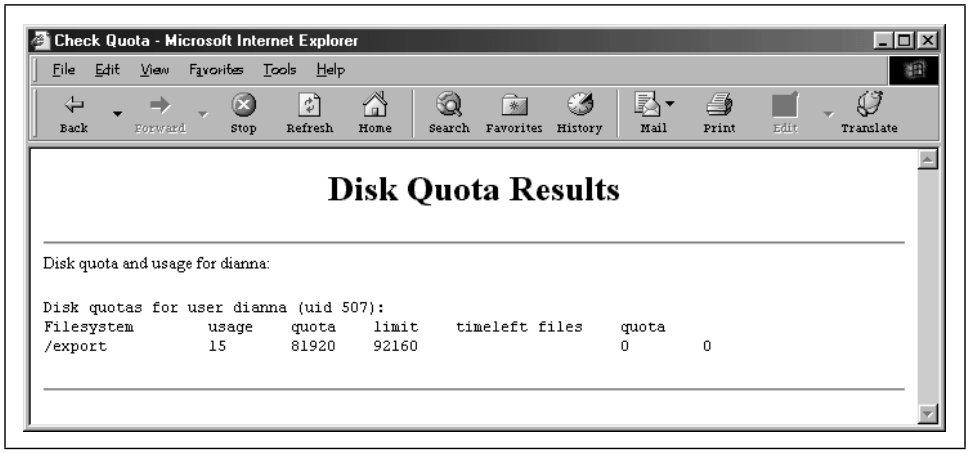

Figure 14-2. Quota CGI form handler results

Checking IMAP quotas

Sites that run the Cyrus IMAP server use quotas specific to the IMAP server, not the Unix operating system. Those sites will use either IMAP itself or the *cyradm* administration utility to report quotas. The next examples show how to check a user's quota using IMAP. The CGI form is shown in Example 14-5.

Example 14-5. CGI Form to Check IMAP Quota

```
#!/usr/local/bin/perl

use CGI;

$query = new CGI;

print $query->header;
print $query->start_html(-title=>'Check IMAP Quota');
print $query->startform(-action=>"imapquota_results.cgi");

print $query->h1($query->center("Check IMAP Quota"));
print $query->hr;
print <<EOM;
<p>Enter your username and password and click the Check Quota button.<p>
EOM
print "Username: ", $query->textfield(-name=>'user'),          $query->br;
print "Password: ", $query->password_field(-name=>'password'), $query->p;
print $query->center($query->submit('action','Check Quota'));
print $query->hr;

print $query->endform;
print $query->end_html;
```

Example 14-6 is the CGI form handler. After authenticating the user, the CGI script connects the user to the IMAP port and issues the IMAP *getquotaroot* directive to get the user's quota. Note that the handler relies on the Telnet.pm module; you may need to install the Net::Telnet module because it is not included with the standard Perl distribution.*

Example 14-6. CGI Form Handler to Check IMAP Quota

```
#!/usr/local/bin/perl

use CGI qw(:standard);
unshift (@INC, '/usr/local/lib/perl5/site_perl/5.005/Net');
use Telnet;

$query = new CGI;
$user     = $query->param('user');
$password = $query->param('password');

print $query->header;
print $query->start_html(-title=>'Check IMAP Quota');
print $query->center($query->h1("IMAP Quota Results"));
print $query->hr;

if (correct_pass("$user","$password") == 1) {

    $quotainfo = check_quota($user,$password);
```

* As of Perl 5.005_03.

Example 14-6. CGI Form Handler to Check IMAP Quota (continued)

```
    print <<EOF;
    IMAP quota and usage for $user:<p>

    <pre>$quotainfo</pre>
EOF
} else {
    print "Login incorrect. Go back and try again."
}

print $query->p,$query->hr;
print $query->end_html;
## Function:    check_quota
##
## Purpose:     Logs user in to IMAP, runs GETQUOTAROOT,
##              returns usage and limit
##
sub check_quota {

    my ($username, $passwd) = @_;
    my $hostname = "imap.unt.edu";

    my $imap = new Net::Telnet (Telnetmode => 0);
    $imap->open(Host => $hostname, Port => 143);

    ## Read the connection message for status

    $line = $imap->getline;
    die $line unless $line =~ /OK/;

    ## Log the user in

    $imap->print("0 login $username $passwd");
    $line = $imap->getline;
    die $line unless $line =~ /OK/;

    ## Get the quota and usage

    $imap->print("0 getquotaroot inbox");
    @lines = $imap->getlines(Timeout => 30);

    foreach $line (@lines) {
        chop $line;

        if ($line =~ /STORAGE/) {
            ($junk, $remainder) = split (/\(/, $line);
            $remainder = substr ($remainder, 0, -1);
            ($resource, $usage, $quota) = split (' ', $remainder);
            last;
        }
    }

    return "Your IMAP quota is $quota Kbytes: usage is $usage Kbytes.\n";
    exit;
}
```

Figure 14-3 and Figure 14-4 are screen shots of the IMAP quota check form and handler results.

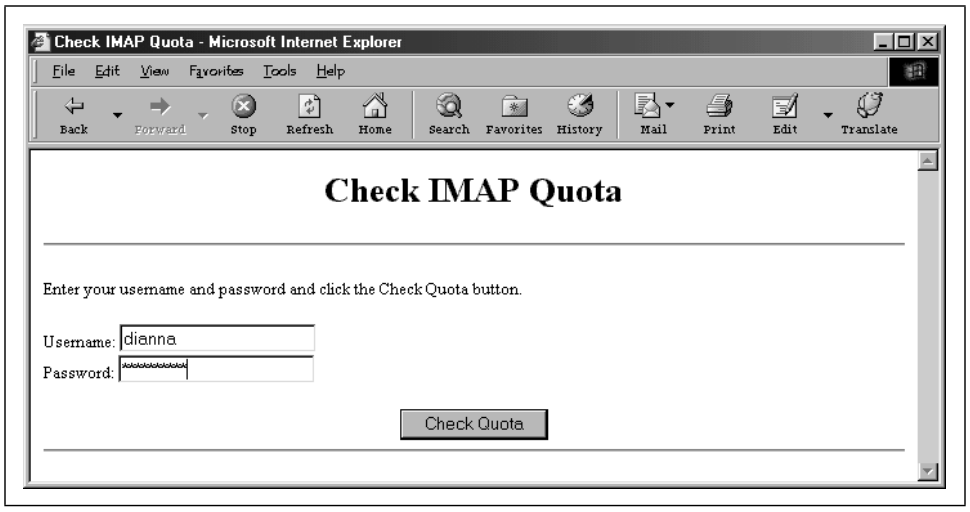

Figure 14-3. IMAP quota CGI form

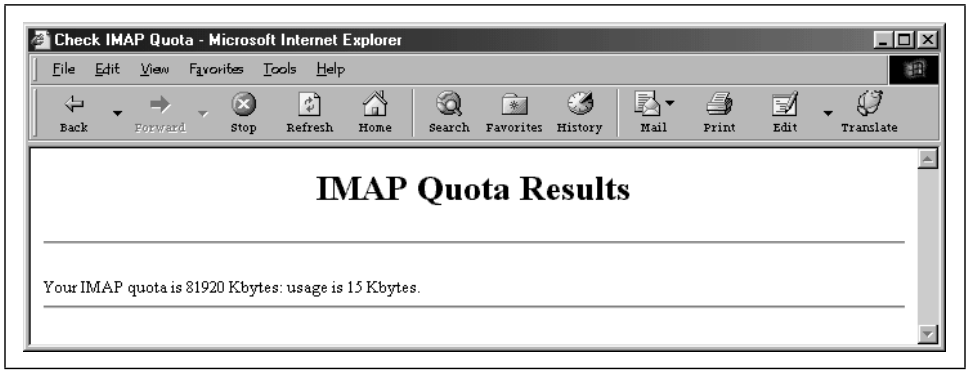

Figure 14-4. IMAP quota CGI form handler results

Mission Restriction

If you decide to bring up dedicated IMAP servers, there's a short list of things you can do to help prepare your host and help focus its activities on the task at hand: IMAP. Primarily, these activities can be grouped together as limiting the number of server processes, eliminating or restricting non-administrative accounts, and reducing non-essential workload on the host.

Reducing Server Processes

Out of the box, your operating system probably supports a variety of services through the *inetd* superserver. None of those services is essential to the IMAP mission. In most cases, you can reduce your *inetd* services down to a single line in your configuration file that supports your particular IMAP service. If non-privileged users never log on to your mail host, you are somewhat freer to make assumptions about what kind of client software those users have. For example, assuming that:

- Interactive logins, if allowed, are done via Secure Shell (SSH), and SSH runs as a standalone daemon, and
- The MTA runs as standalone daemon, as does sendmail

then there's little reason to have anything but a one-line *inetd.conf* file.

Once you've shaved down your *inetd.conf* file, send a HUP signal to it to refresh the active configuration. Then, use *netstat* to get a picture of what kinds of "listens" are still active on your machine. Here's an example from a machine that hasn't completely reduced its *inetd.conf* file yet (the output has been trimmed down with some filtering from *egrep*):

```
% netstat -a | egrep -i '(tcp|udp|listen|\*|local)'

UDP
   Local Address      State
      *.sunrpc        Idle
      *.*             Unbound
      *.32771         Idle
      *.32773         Idle
      *.32774         Idle
      *.tftp          Idle
      *.32776         Idle
      *.lockd         Idle
      *.32779         Idle
      *.syslog        Idle
      *.22370         Idle
      *.nfsd          Idle
      *.32800         Idle
      *.32801         Idle
      *.snmp          Idle
      *.*             Unbound
      *.erpc          Idle
      *.37407         Idle
      *.36161         Idle
      *.36629         Idle
      *.36798         Idle
      *.36799         Idle
      *.36805         Idle
      *.762           Idle
```

```
TCP
   Local Address         Remote Address    Swind Send-Q Rwind Recv-Q  State
      *.sunrpc              *.*               0      0     0      0 LISTEN
      *.32771               *.*               0      0     0      0 LISTEN
      *.imap                *.*               0      0     0      0 LISTEN
      *.cyrus               *.*               0      0     0      0 LISTEN
      *.ftp                 *.*               0      0     0      0 LISTEN
      *.echo                *.*               0      0     0      0 LISTEN
      *.lockd               *.*               0      0     0      0 LISTEN
      *.32772               *.*               0      0     0      0 LISTEN
      *.22370               *.*               0      0     0      0 LISTEN
      *.nfsd                *.*               0      0     0      0 LISTEN
      *.32774               *.*               0      0     0      0 LISTEN
      *.32775               *.*               0      0     0      0 LISTEN
      *.pop-3               *.*               0      0     0      0 LISTEN
      *.7937                *.*               0      0     0      0 LISTEN
      *.3306                *.*               0      0     0      0 LISTEN
      *.80                  *.*               0      0     0      0 LISTEN
      *.smtp                *.*               0      0     0      0 LISTEN
      *.22                  *.*               0      0     0      0 LISTEN
```

Notice that although several of the services are named using the tags from the */etc/services* file, several are not. Also note that *netstat* doesn't tell you which process (for example, *httpd* or *inetd*) is acting as a server in each case.

A much better tool for generating a comprehensive list of services and associated processes running on your host is the *lsof* (list open files) utility. If you've ever used IRIX, you're probably familiar with a similar utility called *fuser*, which has much the same functionality as *lsof*. In addition to showing you the processes that have a given file open, *lsof* can also show you which processes have given TCP and UDP sockets open. The following command produces a comprehensive list of services running on the host, as did the previous *netstat* command, but it also identifies the process and user associated with each service:

```
% lsof -i | egrep -i '(command|listen|idle)' | sort

COMMAND     PID USER   FD   TYPE     DEVICE SIZE/OFF NODE NAME
automount   162 root    9u  inet 0xf5c4e968      0t0  UDP *:762 (Idle)
erpcd       284 root    3u  inet 0xf5f5e650      0t0  UDP *:erpc (Idle)
hpnpd       227 root    3u  inet 0xf5e0f1a0      0t0  UDP *:22370 (Idle)
hpnpd       227 root    4u  inet 0xf5e0f130      0t0  TCP *:22370 (LISTEN)
httpd      2934 root   20u  inet 0xf5e0f2f0      0t0  TCP *:80 (LISTEN)
httpd      4136  www   20u  inet 0xf5e0f2f0      0t0  TCP *:80 (LISTEN)
httpd     17981  www    5u  inet 0xf5e0f280   0t4051  TCP *:57651 (IDLE)
httpd     17981  www   20u  inet 0xf5e0f2f0      0t0  TCP *:80 (LISTEN)
httpd     21277  www   20u  inet 0xf5e0f2f0      0t0  TCP *:80 (LISTEN)
httpd     21425  www   20u  inet 0xf5e0f2f0      0t0  TCP *:80 (LISTEN)
httpd     23094  www   20u  inet 0xf5e0f2f0      0t0  TCP *:80 (LISTEN)
httpd     25665  www   20u  inet 0xf5e0f2f0      0t0  TCP *:80 (LISTEN)
httpd     26498  www   20u  inet 0xf5e0f2f0      0t0  TCP *:80 (LISTEN)
imapd     19062  kwm    6u  inet 0xf5f5e3b0      0t0  UDP *:36798 (Idle)
imapd     19069  kwm    6u  inet 0xf5f5e030      0t0  UDP *:36799 (Idle)
```

```
inetd       136 root    5u  inet 0xf5b302e8      0t0  TCP *:imap (LISTEN)
inetd       136 root    6u  inet 0xf5b30278      0t0  TCP *:cyrus (LISTEN)
inetd       136 root    7u  inet 0xf5b30208      0t0  TCP *:ftp (LISTEN)
inetd       136 root    8u  inet 0xf5b30198      0t0  UDP *:tftp (Idle)
inetd       136 root   12u  inet 0xf5b300b8      0t0  TCP *:echo (LISTEN)
inetd       136 root   21u  inet 0xf5e0f7c0      0t0  TCP *:pop-3 (LISTEN)
lockd       141 root    4u  inet 0xf5e0fc90      0t0  UDP *:lockd (Idle)
lockd       141 root    5u  inet 0xf5e0fc20      0t0  TCP *:lockd (LISTEN)
micq      17882  kwm    5u  inet 0xf5f5e1f0  0t81531  UDP *:36629 (Idle)
mountd      271 root    4u  inet 0xf5f5eb90      0t0  UDP *:32800 (Idle)
mountd      271 root    6u  inet 0xf5f5eab0      0t0  TCP *:32774 (LISTEN)
mysqld     8265 root    3u  inet 0xf5f5e500      0t0  TCP *:3306 (LISTEN)
nfsd        269 root    4u  inet 0xf5f5ed50      0t0  UDP *:nfsd (Idle)
nfsd        269 root    5u  inet 0xf5f5ece0      0t0  TCP *:nfsd (LISTEN)
nscd       5334 root   10u  inet 0xf5e0f3d0      0t0  UDP *:36161 (Idle)
nsrexecd   1377 root    3u  inet 0xf5f5ef10      0t0  TCP *:7937 (LISTEN)
rpc.bootp   276 root    0u  inet 0xf5f5e960      0t0  UDP *:32801 (Idle)
rpc.bootp   276 root    1u  inet 0xf5f5e8f0      0t0  TCP *:32775 (LISTEN)
rpcbind     112 root    3u  inet 0xf5b30d68      0t0  UDP *:sunrpc (Idle)
rpcbind     112 root    5u  inet 0xf5b30c88      0t0  UDP *:32771 (Idle)
rpcbind     112 root    6u  inet 0xf5b30c18      0t0  TCP *:sunrpc (LISTEN)
rpcbind     112 root    7u  inet 0xf5b30ba8      0t0  TCP *:61409 (IDLE)
sendmail  22010 root    7u  inet 0xf5e0f440      0t0  TCP *:smtp (LISTEN)
snmpd       279 root    0u  inet 0xf5f5e730      0t0  UDP *:snmp (Idle)
sshd        370 root    3u  inet 0xf5c4e658      0t0  TCP *:22 (LISTEN)
sshd      19112 root    8u  inet 0xf5f5e180      0t0  UDP *:36805 (Idle)
statd       139 root    3u  inet 0xf5e0fe50      0t0  UDP *:32776 (Idle)
statd       139 root    4u  inet 0xf5e0fd00      0t0  TCP *:32772 (LISTEN)
statd       139 root    9u  inet 0xf5e0f670      0t0  UDP *:32779 (Idle)
statd       139 root   10u  inet 0xf5c4ef18      0t0  UDP *:37407 (Idle)
syslogd     166 root    4u  inet 0xf5e0f4b0      0t0  UDP *:syslog (Idle)
ypbind      120 root    4u  inet 0xf5b30518      0t0  UDP *:32773 (Idle)
ypbind      120 root    6u  inet 0xf5b30588      0t0  UDP *:32774 (Idle)
ypbind      120 root   10u  inet 0xf5b304a8      0t0  TCP *:32771 (LISTEN)
```

This listing shows the actual command, process ID, user, and TCP or UDP port associated with each service running on the current host. Once you have the actual command and user name, you can find out how the service gets started and whom to contact about moving a service to another machine, if necessary.

The next order of business would be to examine the users on your host and eliminate as many as possible, or at least remove the ability to log in interactively. With the increasing popularity of NIS, Kerberos, and the variety of authentication methods usable with the PAM interface, it's possible that there are many more users of your mail host than those explicitly listed in your */etc/passwd* file. Users may be listed in a number of places, including a NIS password map or a CRAM password database. PAM configuration files (on many systems, under */etc/pam.d/*) may also have clues as to where authentication credentials are defined for your users.

Finally, run the *ps* command on your machine, audit the *crontabs*, *at* queues (usually in */var/spool/cron*), and */etc/rc?.d/** files to get a good handle on not only what

is currently running on your host, but what is likely to run each time you start it and at any arbitrary point in the future. Your first reaction to this recommendation might be to dismiss it because, after all, you're the system administrator of your mail server and you ought to know everything that goes on. If more than one person has root access to your server, however, no matter how finely tuned your workflow is, you're likely to find at least a few subtle surprises when you do an audit of your host.

The Ultimate in Dedicated Servers

We want to briefly mention the concept of separating SMTP from IMAP; that is, using separate machines to perform IMAP access to the mailstore and SMTP routing. In this separate server scheme, the SMTP router uses Local Mail Transport Protocol (LMTP) to talk to the IMAP server, as shown in Figure 14-5.

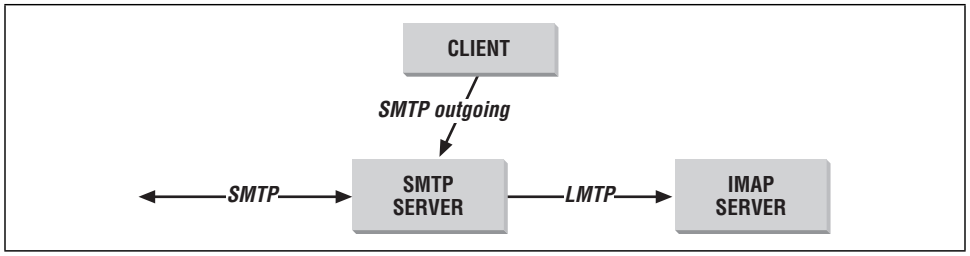

Figure 14-5. Separate SMTP and IMAP servers

That way, the IMAP server doesn't spend cycles and I/O bandwidth on managing an SMTP queue. This scheme is already being used successfully by a few early adopters and is beginning to be used more widely.