CHAPTER 7

E-Mailing with CGI

OVERVIEW

One of the most important tasks of any CGI program is ultimately to let someone know that something has happened. The most convenient way for users is to have this done automatically using E-mail. In this chapter we will examine two mechanisms for sending E-mail.

The first method uses **SENDMAIL-LIB.PL**, a library that was written to allow a programmer to use the sendmail program on UNIX to send E-mail easily. This is the most reliable and safest way to send E-mail using a CGI script on UNIX.

Unfortunately, using the sendmail program is a UNIX-specific option. If you want your scripts to run on Windows NT or another non-UNIX platform, you will have to use another option, such as the Windows NT–specific BLAT routines. Another alternative is the **SMTPMAIL-LIB.PL** library,

which sends mail on either Windows NT or UNIX. This library uses TCP/IP sockets to communicate directly with simple mail transfer protocol (SMTP) servers to send mail. We will explain the concepts of sockets and the SMTP protocol later in this chapter.

Internet mail is notoriously insecure. Internet mail is sent as "cleartext" over the Internet. In other words, the packets that make up an Internet mail letter can be seen by anyone who has physical access to the network and the proper tools much like a person with the appropriate radio frequency scanner can listen in on cellular phone calls.

For example, with the shopping cart application, E-mail is not the best mechanism to use for sending information over the Internet especially when dealing with credit cart numbers submitted with an order.

A more secure way to send e-mail is to make sure that the CGI program is sending it to an account on the same machine or local network that the E-mail address is on. That way, the Internet mail never actually has to be transferred outside of the confines of your servers. If you need a secure way of sending E-mail to the outside world, you should look at programs such as PGP (Pretty Good Privacy) to encrypt your e-mail before using this library.

Installation and Usage

Both **SENDMAIL-LIB.PL** and **SMTPMAIL-LIB.PL** are designed to be used in exactly the same way. You can `require` one of them in a program; then, if you change your mind about the method of E-mailing, you simply use the other library. To use either one, you copy the library of choice over the filename **mail-lib.pl**. All of our CGI programs that rely on E-mail `require` **mail-lib.pl** as a standard mailing file. By copying over the appropriate library, you are making a choice as to which method to use (see Figure 7.1).

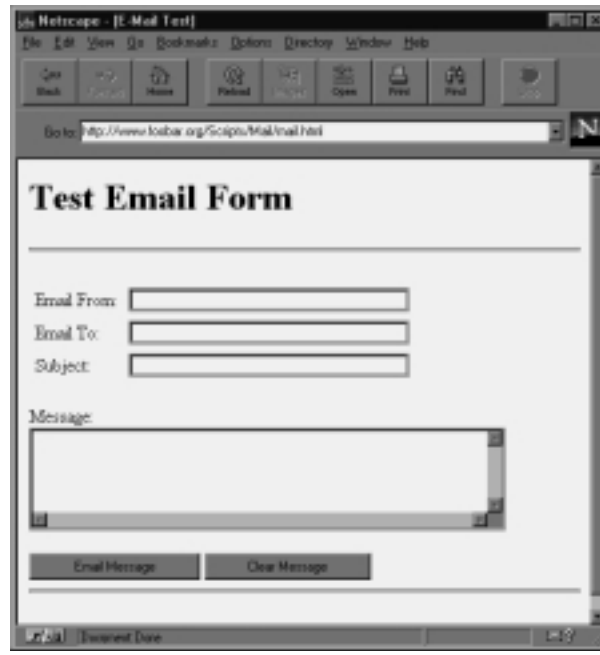

Figure 7.1 *Sample of using* ***MAIL-LIB.PL*** *in a CGI program.*

Sending Email

Both libraries have a `send_mail` function. Sending mail consists of passing the "From" E-mail address, the "To" E-mail address, the message subject, and the message body as parameters to the function. Here is an example of sending mail, with smith@zzz.org sending E-mail to jones@yyy.com:

```
&send_mail("smith\@zzz.org", "jones\@yyy.org",
  "Subject", "This is the message\nto send.\n\n");
```

Note that we escaped the `@` symbol with a backslash. This arrangement is necessary for the code to operate correctly in Perl version 5, which requires that `@` symbols be escaped.

Sending Email with More Options

There is also a `real_send_mail` function in each library. This function accepts the same basic parameters as `send_mail` except that it breaks up the "From" and "To" addresses into E-mail address, hostname pairs. Here is the equivalent `real_send_mail` version of the previous example:

```
&real_send_mail("smith\@zzz.org", "zzz.org",
  "jones\@yyy.org", "yyy.org", "Subject",
  "This is the message\nto send.\n\n");
```

It may seem weird that we have a function that does the same thing as `send_mail` but is harder to call because we must use more parameters. It is important to understand that `send_mail` consists of code that breaks apart the "From" and "To" addresses and separates them into parameters that it uses to call `real_send_mail`. In both libraries, the `real_send_mail` function is the actual mailer function. `Send_mail` is provided as an easier way to send the E-mail.

If you are using the SENDMAIL version of the library, you probably won't need to call `real_send_mail` directly. However, if you are using the SMTPMAIL version, you may need to use `real_send_mail`, because some servers use surrogate hosts to handle their E-mail. When you send mail to an address such as yyy.org, it is quite possible that yyy.org does not handle its own E-mail; instead it may use another host, such as mail.yyy.org to handle E-mail destined for yyy.org. The Internet tries to handle this by setting up MX (mail exchanger) records as part of the domain name resolution protocol. A host such as yyy.org would have an MX record set aside to tell the name lookup routine to return mail.yyy.org as the actual host to mail to. The UNIX sendmail program handles this problem automatically and resolves MX records transparently to the user. Unfortunately, because the SMTP version of the mail library communicates directly between SMTP mail hosts, it does not go through the MX record resolution.

When to Use the SMTP-MAIL.PL Library

Why would we want to use the SMTP library? It may sound as if the SMTP library is not very useful, but there are several cases in which you may still want to use it. For one thing, the SENDMAIL library operates only on UNIX. The SMTP library can be used on Windows NT and any other platform that supports sockets programming in Perl. Also, SENDMAIL cannot change the true From: address of the E-mail message. Because the SMTP library communicates using sockets, it can fool an SMTP server into thinking you are any user on that system; sometimes it is nice not to have mail being sent as wwwadmin or "nobody" to your users. Finally, we can get around the MX problem by calling the `real_send_mail` function, and, in place of the hostname to send mail to, we use the MX hostname if you happen to know it. Using the example where mail.yyy.org handles the E-mail for yyy.org, we could send mail using the following snippet of code:

```
&real_send_mail("smith\@zzz.org", "zzz.org",
"jones\@yyy.org", "mail.yyy.org", "Subject",
"This is the message\nto send.\n\n");
```

In addition, **SMTPMAIL-LIB.PL** has a global operating system variable called `$smtp_os`. By default, this variable is set to "UNIX." If you are using an NT server version of Perl that does not support the `SELECT()` command, you need to set this variable equal to "NT." In addition, if you are using a version of Perl that does not support the sockets package (**Socket.pm**), remove the line in the program that says `use Socket` and replace the `$AF_INET` and `$SOCK_STREAM` variable assignment values with 2 and 1, respectively. On some systems, these values may be different. For example, on Solaris UNIX, `$SOCK_STREAM` should be set to 2.

For the most part, you will probably want to use sendmail; it's reliable and well tested and handles many things, such as MX record resolution, transparently. However, if you need to do something that only the SMTP

version of the library can handle, you always have the option of using it by copying it over the **mail-lib.pl** library program and calling the same routines you would if you had used the sendmail version.

Design Discussion

SENDMAIL-LIB.PL

SENDMAIL-LIB operates by using the sendmail program on UNIX to send the E-mail. Why do we use sendmail instead of some other UNIX program such as mail? The reason lies with security. Because a lot of mail that gets sent via a CGI script is sent to an address that was entered into an HTML form, it is at best precarious to pass user input as a command-line parameter to a program such as the UNIX mail program. However, the UNIX sendmail program allows us to pass the From:, To:, Subject:, and message body text as part of the `STDIN` input stream of the program when opened using the `-t` command-line parameter. In that way, no user form input actually gets passed as a command-line parameter to the UNIX-based Web server. In addition to using the `-t` flag, we call sendmail using the `-n` flag. This tells the sendmail program not to use the alias list of the server. Most servers consider the alias list to be somewhat protected information. The `$mail_program` variable is set to the path of the sendmail program plus the command-line parameters. If, for some reason, you wish the alias list to be used, you can take out the `-n` parameter.

```
$mail_program = "/usr/lib/sendmail -t -n";
```

REAL_SEND_MAIL Subroutine

The `real_send_mail` performs the sending process. The `send_mail` subroutine described later in this chapter calls `real_send_mail` after parsing its own input parameters.

`Real_send_mail` accepts as parameters the "From" E-mail address and SMTP server hostname, the "To" address and SMTP server hostname,

and the subject and message body. Some of these parameters may not be used within the real_send_mail function in this library, but we keep the parameters to maintain compatibility with the SMTP version.

```
sub real_send_mail {
    local($fromuser, $fromsmtp, $touser, $tosmtp,
      $subject, $messagebody) = @_;
```

First, we need to start the mail program and open a pipe to the program so that anything we print to the file handle, MAIL, gets output to the running program.

```
    open (MAIL, "|$mail_program") ||
     &web_error("Could Not Open Mail Program");
```

The rest of the function is relatively easy. Merely print the header information to the $mail_program that is currently running and then close the file handle to the program. Closing the file handle triggers the sendmail program on UNIX to send the mail. We use the HERE document method to print the message to the sendmail program between the __END_OF_MAIL__ tags.

```
print MAIL <<__END_OF_MAIL__;
To: $touser
From: $fromuser
Subject: $subject

$messagebody

__END_OF_MAIL__

    close (MAIL);

} #end of real_send_mail
```

SEND_MAIL SUBROUTINE

The send_mail function as parameters uses the "From" and "To" address as well as the subject and message body. It takes the "From" and "To" addresses and splits them up into E-mail address, hostname pairs, which are sent to real_send_mail, the routine that actually does the work of sending the E-mail.

```
sub send_mail {
    local($from, $to, $subject, $messagebody) = @_;
    local($fromuser, $fromsmtp, $touser, $tosmtp);
```

`$fromuser` and `$touser` are equal to the actual E-mail addresses that were passed before.

```
    $fromuser = $from;
    $touser = $to;
```

`Split` is used to break the address into user and hostname pairs. The hostname is the second element of the split array, so we reference the hostname with a 1 (arrays start at 0). We can do this because the output of the split command is an array; we reference the output of `split` as if it were elements of an array.

```
    $fromsmtp = (split(/\@/,$from))[1];
    $tosmtp = (split(/\@/,$to))[1];
```

Finally, we call the `real_send_mail` subroutine.

```
&real_send_mail($fromuser, $fromsmtp, $touser,
           $tosmtp, $subject, $messagebody);

} # End of send_mail
```

web_error Subroutine

`web_error` is a subroutine whose purpose is to print errors as HTML before exiting the program. Most Perl programs use the `DIE` subroutine to exit the program prematurely. Unfortunately, using `DIE` typically does not print any useful information to the user attempting to set up the script. Thus, we use `web_error` to print the message as HTML and then call `DIE`. The `DIE` subroutine in Perl is still useful for a Web server, because the message given by `DIE` is generally printed to the Web server's error log.

```
sub web_error {
    local ($error) = @_;
    $error = "Error Occured: $error";
```

```
    print "$error<p>\n";

# Die exits the program prematurely and
# prints an error to stderr

    die $error;

} # end of web_error
```

SMTPMAIL-LIB.PL

SMTPMAIL-LIB.PL works basically the same as **SENDMAIL-LIB.PL** except that the SMTP version communicates directly to mail servers to send mail from one person to another instead of using a prepackaged program such as the UNIX sendmail program. In this section, we will describe the socket-specific functions in detail and go over the other functions briefly, because they have same explanation as those in the **SENDMAIL-LIB.PL** library.

SMTPMAIL-LIB does all its work by using sockets programming. Basically, sockets can be thought of as synonymous with Telnet, except that instead of using Telnet to connect to a UNIX server and run programs, you are using Telnet to connect to a service such as an E-mail service (SMTP, POP) or an HTML service (HTTP). Whenever an E-mail package sends E-mail over the Internet, it ultimately connects to a mail server. In other words, it plugs itself into the mail server's "socket." SMTPMAIL-LIB operates by connecting to a mail server, sending the SMTP mail information, and then disconnecting from the mail server.

By default, we use the **Socket.pm** package in Perl 5. If you are not using Perl 5 or lack the **Socket.pm** file, you can delete this line and set the `$AF_INET` and `$SOCK_STREAM` variables manually in the Perl program according to the instructions in the usage section of this chapter. `$http_os` should be set to "NT" if you are using a Windows NT version of Perl. `$http_os` is set to "UNIX" by default.

REAL_SEND_MAIL SUBROUTINE

The `real_send_mail` function in SMTPMAIL-LIB has the same parameters as the `real_send_mail` in SENDMAIL-LIB. However, the `real_send_mail` function in SMTPMAIL-LIB operates by opening a socket connection to an SMTP server.

```
sub real_send_mail {
    local($fromuser, $fromsmtp, $touser, $tosmtp,
       $subject, $messagebody) = @_;
    local($ipaddress, $fullipaddress, $packconnectip);
    local($packthishostip);
    local($AF_INET, $SOCK_STREAM, $SOCK_ADDR);
    local($PROTOCOL, $SMTP_PORT);
    local($buf);
# We start off by making the message that will be sent
# By combining the subject with the message body text
#
    $messagebody = "Subject: $subject\n\n" . $messagebody;
```

The following variables are set using values defined in the **Socket.pm** library. If your version of Perl does not have the sockets library, you can substitute default values, such as 2 for `AF_INIT` and 1 for `SOCK_STREAM`. If 1 does not work for `SOCK_STREAM`, try using 2. `$SOCK_ADDR` is a special variable that holds the IP address, port, and other relevant socket information.

```
    $AF_INET = AF_INET;
    $SOCK_STREAM = SOCK_STREAM;

    $SOCK_ADDR = "S n a4 x8";

# The following routines get the protocol information

    $PROTOCOL = (getprotobyname('tcp'))[2];
    $SMTP_PORT = (getservbyname('smtp','tcp'))[2];

    $SMTP_PORT = 25 unless ($SMTP_PORT =~ /^\d+$/);
    $PROTOCOL = 6 unless ($PROTOCOL =~ /^\d+$/);
```

We call the `gethostbyname` function to convert the hostname we are connecting to into a real IP address. `$fullipaddress` is the ASCII equivalent of the IP address delimited with periods. We calculate this in case we want to troubleshoot whether the hostname lookup actually worked. `$packconnectip` is the host to connect to, given in the special `$SOCK_ADDR` format. `$packthishostip` is also in the `$SOCK_ADDR` format and designates this local host.

```
    $ipaddress = (gethostbyname($tosmtp))[4];

    $fullipaddress
```

```
      = join (".", unpack("C4", $ipaddress));

    $packconnectip = pack($SOCK_ADDR, $AF_INET,
             $SMTP_PORT, $ipaddress);
    $packthishostip = pack($SOCK_ADDR,
                $AF_INET, 0, "\0\0\0\0");
```

The socket first needs to be created. Creating the socket is very much like opening a filename for writing except that it accepts information about the protocol it is using instead of a filename. In fact, "S" is actually a filehandle to and normal file operations can be performed on the socket.

Then, the program binds the socket to the local host (`$packthishostip`) using the bind command. This is a necessary extra housekeeping step just to let the socket know where to send the data back to. This may seem like a silly extra step, but keep in mind that a machine may have multiple network cards in it and have multiple IP addresses. Thus, it is generally good to explicitly bind the socket to one of those local addresses.

Finally, the script connects the bound socket using the connect statement to the address designated by `$packconnectip`. This is the part that actually connects the Web server's machine over the Internet to the mail SMTP server.

```
socket (S, $AF_INET, $SOCK_STREAM, $PROTOCOL) ||
     &web_error( "Can't make socket:$!\n");

bind (S,$packthishostip) ||
     &web_error( "Can't bind:$!\n");

connect(S, $packconnectip) ||
     &web_error( "Can't connect socket:$!\n");
```

We select the socket handle and then issue the `$| = 1` command to turn off buffering. We want all commands that get transmitted to the socket to be sent right away. `STDOUT` is then selected as the current default output. The handle to the socket is returned to the routine that called `OpenSocket` so that other routines can use the socket.

```
    select(S);
    $| = 1;
    select (STDOUT);
```

Now the mail message is sent to the mail server. First, we read the connection line from the SMTP server we are connecting to. Then we send the SMTP mail commands one by one.

```
$buf = read_sock(S, 6);
```

The `HELO` command establishes a connection between the SMTP server we are on and the SMTP server from which we are supposed to be sending E-mail. We need not be connected to an E-mail server; it can refer to any SMTP server on the Internet. Because SMTP is an insecure protocol, you can send mail from any server to any server on the Internet and pretend you are someone else. Note that I do not recommend doing this for the purpose of deception. This library was created so that you can send E-mail without having the "Nobody" user ID show up as part of the E-mail message that the Web server sends.

```
print S "HELO $fromsmtp\n";

$buf = read_sock(S, 6);
```

The `MAIL` command sets up the "From: username" part of the E-mail.

```
print S "MAIL From:<$fromuser>\n";
$buf = read_sock(S, 6);
```

The `RCPT` sets up the "To: username" part.

```
print S "RCPT To:<$touser>\n";
$buf = read_sock(S, 6);
```

The `DATA` flags the SMTP server to start accepting the message as free-form input. This input is terminated by ending the line with a single period followed by a newline.

```
print S "DATA\n";
$buf = read_sock(S, 6);

print S $messagebody . "\n";
```

The period followed by a newline finishes the message and sends it. We send a `QUIT` statement to the server to log off, and we close the socket.

```
    print S ".\n";
    $buf = read_sock(S, 6);

    print S "QUIT\n";

    close S;
```

} #END OF REAL_SEND_MAIL

send_mail Subroutine

The `send_mail` routine in **SMTPMAIL-LIB.PL** operates exactly the same way as the same routine in **SENDMAIL-LIB.PL** described earlier.

```
sub send_mail
{
local($from, $to, $subject, $messagebody) = @_;

local($fromuser, $fromsmtp, $touser, $tosmtp);

$fromuser = $from;
$touser = $to;

$fromsmtp = (split(/\@/,$from))[1];
$tosmtp = (split(/\@/,$to))[1];

&real_send_mail($fromuser, $fromsmtp, $touser,
           $tosmtp, $subject, $messagebody);

} # End of send_mail
```

read_sock Subroutine

The `read_sock` command reads the next 1024 characters waiting to be read in the socket. It gets passed a handle to the socket to be read, along with a timeout value. The timeout value is listed in seconds. If there is no

input on read_sock in that many seconds, read_sock returns with nothing in the read buffer.

```
sub read_sock {
    local($handle, $endtime) = @_;
    local($localbuf,$buf);
    local($rin,$rout,$nfound);
```

$endtime is the timeout. We set $endtime to $endtime + time and later keep looping until time is greater than $endtime + the original time that the read_sock was called. Additionally, we clear the buffer.

```
# Set endtime to be time + endtime.
    $endtime += time;

# Clear buffer
    $buf = "";
```

$rin is set to be a vector of the socket file handle. This Perl code sets up $rin as input to a special form of the select() function that checks whether input is waiting to be read from the file handle. Because the read command blocks the operating system from doing anything else if there is nothing to read, it makes sense that we want to check for this condition first.

Note that the Windows NT version of Perl does not support the select statement in this form, so we read the socket even if it happens to block us on Windows NT.

```
     $rin = '';
# Set $rin to be a vector of the socket file handle
    vec($rin, fileno($handle), 1) = 1;

# nfound is 0 since we have not read anything yet
    $nfound = 0;
```

The following code loops until there is something to read or we time out. If there is something to read, then $nfound is equal to the number of bytes waiting to be read.

```
read_socket:
while (($endtime > time) && ($nfound <= 0)) {
# Read 1024 bytes at a time
    $length = 1024;
# Preallocate buffer
    $localbuf = " " x 1025;
     # NT does not support select for polling to see if
     # There are characters to be received.
     # This is important Because we don't want to block
     # if there is nothing being received.
    $nfound = 1;
    if ($mail_os ne "NT") {
# The following polls to see if there is anything in
# the input buffer to read.  If there is, we will later
# call the sysread routine
     $nfound = select($rout=$rin, undef, undef,.2);
         }
}
```

If we have found characters waiting to be read as part of the `$nfound` command, we read the socket using the `sysread` command. Finally, the contents of the buffer are returned to the routine that called the `read_sock` function.

```
     if ($nfound > 0) {
     $length = sysread($handle, $localbuf, 1024);
     if ($length > 0) {
         $buf .= $localbuf;
         }
    }

$buf;
}
```

web_error Subroutine

There is nothing new here. This web_error subroutine is the same one you saw previously explained with sendmail-lib.pl.

```
sub web_error
{
local ($error) = @_;
```

```
$error = "Error Occured: $error";
print "$error<p>\n";

# Die exits the program prematurely
# and prints an error to stderr

die $error;

} # end of web_error
```