CHAPTER 16

Networking

You may have turned to this page wondering just what PHP could possibly have to offer in regards to networking. After all, aren't networking tasks largely relegated to languages commonly used for system administration, such as Perl or Python? While such a stereotype might have once painted a fairly accurate picture, these days, incorporating networking capabilities into a Web application is commonplace. In fact, Web-based applications are regularly used to monitor and even maintain network infrastructures. Furthermore, with the introduction of the command-line interface (CLI) in PHP version 4.2.0, PHP is now increasingly used for system administration among those developers who wish to continue using their favorite language for other purposes. The PHP developers, always keen to acknowledge growing needs in the realm of Web application development, and remedy that demand by incorporating new features into the language, have put together a rather amazing array of network-specific functionality.

This chapter is divided into several topics, each of which is previewed here:

- **DNS, servers, and services:** PHP offers a variety of functions capable of retrieving information about the internals of networks, DNS, protocols, and Internet addressing schemes. This chapter introduces these functions and offers several usage examples.
- **Sending e-mail with PHP:** Sending e-mail via a Web application is undoubtedly one of the most commonplace features you can find these days, and for good reason. E-mail remains the Internet's killer application, and offers an amazingly efficient means for communicating and maintaining important data and information. This chapter explains how to effectively imitate even the most proficient e-mail client's "send" functionality via a PHP script.
- **IMAP, POP3, and NNTP:** PHP's IMAP extension is, despite its name, capable of communicating with IMAP, POP3, and NNTP servers. This chapter introduces many of the most commonly used functions found in this library, showing you how to effectively manage an IMAP account via the Web.
- **Streams:** Introduced in version 4.3, streams offer a generalized means for interacting with *streamable* resources, or resources that are read from and written to in a linear fashion. This chapter offers an introduction to this feature.
- **Common networking tasks:** To wrap up this chapter, you'll learn how to use PHP to mimic the tasks commonly carried out by command-line tools, including pinging a network address, tracing a network connection, scanning a server's open ports, and more.

DNS, Services, and Servers

These days, investigating or troubleshooting a network issue often involves gathering a variety of information pertinent to affected clients, servers, and network internals such as protocols, domain name resolution, and IP addressing schemes. PHP offers a number of functions for retrieving a bevy of information about each subject, each of which is introduced in this section.

DNS

The DNS is what allows us to use domain names (example.com, for instance) in place of the corresponding not-so-user-friendly IP address, such as 192.0.34.166. The domain names and their complementary IP addresses are stored and made available for reference on domain name servers, which are interspersed across the globe. Typically, a domain has several types of records associated to it, one mapping the IP address to the domain, another for directing e-mail, and another for a domain name alias, for example. Often, network administrators and developers require a means to learn more about various DNS records for a given domain. This section introduces a number of standard PHP functions capable of digging up a great deal of information regarding DNS records.

checkdnsrr()

```
int checkdnsrr (string host [, string type])
```

The `checkdnsrr()` function checks for the existence of DNS records based on the supplied `host` value and optional DNS resource record `type`, returning `TRUE` if any records are located and `FALSE` otherwise. Possible record types include the following:

- **A:** IPv4 Address Record. Responsible for the hostname-to-IPv4 address translation.
- **AAAA:** IPv6 Address Record. Responsible for the hostname-to-IPv6 address translation.
- **A6:** A record type used to represent IPv6 addresses. Intended to supplant present use of AAAA records for IPv6 mappings.
- **ANY:** Looks for any type of record.
- **CNAME:** Canonical Name Record. Maps an alias to the real domain name.
- **MX:** Mail Exchange Record. Determines the name and relative preference of a mail server for the host. This is the default setting.
- **NAPTR:** Naming Authority Pointer. Used to allow for non-DNS-compliant names, resolving them to new domains using regular expression rewrite rules. For example, an NAPTR might be used to maintain legacy (pre-DNS) services.
- **NS:** Name Server Record. Determines the name server for the host.
- **PTR:** Pointer Record. Used to map an IP address to a host.

- **SOA:** Start of Authority Record. Sets global parameters for the host.
- **SRV:** Services Record. Used to denote the location of various services for the supplied domain.

Consider an example. Suppose you want to verify whether the domain name `example.com` has been taken:

```
<?php
   $recordexists = checkdnsrr("example.com", "ANY");
   if ($recordexists) echo "The domain name has been taken. Sorry!";
   else echo "The domain name is available!";
?>
```

This returns the following:

```
The domain name has been taken. Sorry!
```

You can use this function to verify the existence of a domain of a supplied mail address:

```
<?php
   $email = "ceo@example.com";
   $domain = explode("@",$email);

   $valid = checkdnsrr($domain[1], "ANY");

   if($valid) echo "The domain has an MX record!";
   else echo "Cannot locate MX record for $domain[1]!";
?>
```

This returns:

```
The domain has an MX record!
```

Note that this isn't a request for verification of the existence of an MX record. Sometimes network administrators employ other configuration methods to allow for mail resolution without using MX records (because MX records are not mandatory). To err on the side of caution, just check for the existence of the domain, without specifically requesting verification of whether an MX record exists.

dns_get_record()

```
array dns_get_record (string hostname [, int type
                     [, array &authns, array &addtl]])
```

The dns_get_record() function returns an array consisting of various DNS resource records pertinent to the domain specified by hostname. Although by default dns_get_record() returns all records it can find specific to the supplied domain, you can streamline the retrieval process by specifying a type, the name of which must be prefaced with DNS_. This function supports all the types introduced along with checkdnsrr(), in addition to others that will be introduced in a moment. Finally, if you're looking for a full-blown description of this hostname's DNS description, you can pass the authns and addtl parameters in by reference, which specify that information pertinent to the authoritative name servers and additional records also should be returned.

Assuming that the supplied hostname is valid and exists, a call to dns_get_record() returns at least four attributes:

- host: Specifies the name of the DNS namespace to which all other attributes correspond.
- class: Because this function only returns records of class "Internet," this attribute always reads IN.
- type: Determines the record type. Depending upon the returned type, other attributes might also be made available.
- ttl: The record's time-to-live, calculating the record's original TTL minus the amount of time that has passed since the authoritative name server was queried.

In addition to the types introduced in the section on checkdnsrr(), the following domain record types are made available to dns_get_record():

- **DNS_ALL:** Retrieves all available records, even those that might not be recognized when using the recognition capabilities of your particular operating system. Use this when you want to be absolutely sure that all available records have been retrieved.
- **DNS_ANY:** Retrieves all records recognized by your particular operating system.
- **DNS_HINFO:** A host information record, used to specify the operating system and computer type of the host. Keep in mind that this information is not required.
- **DNS_NS:** A name server record, used to determine whether the name server is the authoritative answer for the given domain, or whether this responsibility is ultimately delegated to another server.

To forego redundancy, the preceding list doesn't include the types already introduced along with checkdnsrr(). Keep in mind that those types are also available to dns_get_record(). Just remember that the type names must always be prefaced with DNS_.

Consider an example. Suppose you want to learn more about the example.com domain:

```
<?php
   $result = dns_get_record("example.com");
   print_r($result);
?>
```

A sampling of the returned information follows:

```
Array (
   [0] => Array (
      [host] => example.com
      [type] => NS
      [target] => a.iana-servers.net
      [class]  =>  IN
      [ttl]  => 110275
      )
   [1] => Array (
      [host] => example.com
      [type] => A
      [ip] => 192.0.34.166
      [class] => IN
      [ttl] => 88674
      )
)
```

If you were only interested in the name server records, you could execute the following:

```
<?php
   $result = dns_get_record("example.com","DNS_CNAME");
   print_r($result);
?>
```

This returns the following:

```
Array ( [0] => Array ( [host] => example.com [type] => NS
[target] => a.iana-servers.net [class] => IN [ttl] => 21564 )
[1] => Array ( [host] => example.com [type] => NS
[target] => b.iana-servers.net [class] => IN [ttl] => 21564 ) )
getmxrr()
```

getmxrr()

```
int getmxrr (string hostname, array &mxhosts [, array &weight])
```

The `getmxrr()` function retrieves the MX records for the host specified by `hostname`. The MX records are added to the array specified by `mxhosts`. If the optional input parameter `weight` is supplied, the corresponding weight values will be placed there, which refer to the hit prevalence assigned to each server identified by record. An example follows:

```
<?php
   getmxrr("wjgilmore.com",$mxhosts);
   print_r($mxhosts);
?>
```

This returns the following:

```
Array ( [0] => mail.wjgilmore.com)
```

Services

Although we often use the word "Internet" in a generalized sense, making statements pertinent to using the Internet to chat, read, or download the latest version of some game, what we're actually referring to is one or several Internet services that collectively define this communications platform. Examples of these services include HTTP, FTP, POP3, IMAP, and SSH. For various reasons (an explanation of which is beyond the scope of this book), each service commonly operates on a particular communications port. For example, HTTP's default port is 80, and SSH's default port is 22. These days, the widespread need for firewalls at all levels of a network makes knowledge of such matters quite important. Two PHP functions, `getservbyname()` and `getservbyport()`, are available for learning more about services and their corresponding port numbers.

getservbyname()

```
int getservbyname (string service, string protocol)
```

The `getservbyname()` function returns the port number of the service corresponding to `service` as specified by the `/etc/services` file. The `protocol` parameter specifies whether you're referring to the `tcp` or `udp` component of this service. Consider an example:

```
<?php
   echo "HTTP's default port number is: ".getservbyname("http", "tcp");
?>
```

This returns the following:

```
HTTP's default port number is: 80
```

getservbyport()

```
string getservbyport (int port, string protocol)
```

The `getservbyport()` function returns the name of the service corresponding to the supplied `port` number as specified by the `/etc/services` file. The `protocol` parameter specifies whether you're referring to the `tcp` or `udp` component of the service. Consider an example:

```
<?php
   echo "Port 80's default service is: ".getservbyport(80, "tcp");
?>
```

This returns the following:

```
Port 80's default service is: http
```

Establishing Socket Connections

In today's networked environment, you'll often want to query services, both local and remote. Often this is done by establishing a socket connection with that service. This section demonstrates how this is accomplished, using the `fsockopen()` function.

fsockopen()

```
resource fsockopen (string target, int port [, int errno [, string errstring
                   [, float timeout]]])
```

The `fsockopen()` function establishes a connection to the resource designated by `target` on port `port`, returning error information to the optional parameters `errno` and `errstring`. The optional parameter `timeout` sets a time limit, in seconds, on how long the function will attempt to establish the connection before failing.

The first example shows how to establish a port 80 connection to `www.example.com` using `fsockopen()` and how to output the index page:

```
<?php

   // Establish a port 80 connection with www.example.com
   $http = fsockopen("www.example.com",80);

   // Send a request to the server
   $req = "GET / HTTP/1.1\r\n";
   $req .= "Host: www.example.com\r\n";
   $req .= "Connection: Close\r\n\r\n";

   fputs($http, $req);

   // Output the request results
   while(!feof($http))
   {
      echo fgets($http, 1024);
   }

   // Close the connection
   fclose($http);

?>
```

This returns the following:

```
HTTP/1.1 200 OK Date: Mon, 05 Jan 2006 02:17:54 GMT Server: Apache/1.3.27 (Unix)
(Red-Hat/Linux) Last-Modified: Wed, 08 Jan 2006 23:11:55 GMT ETag:
"3f80f-1b6-3e1cb03b" Accept-Ranges: bytes Content-Length: 438
Connection: close Content-Type: text/html
You have reached this web page by typing "example.com", "example.net", or
"example.org" into your web browser.
These domain names are reserved for use in documentation and are not available
for registration. See RFC 2606, Section 3.
```

The second example, shown in Listing 16-1, demonstrates how to use `fsockopen()` to build a rudimentary port scanner.

Listing 16-1. *Creating a Port Scanner with fsockopen()*

```
<?php

   // Give the script enough time to complete the task
   ini_set("max_execution_time", 120);

   // Define scan range
   $rangeStart = 0;
   $rangeStop = 1024;

   // Which server to scan?
   $target = "www.example.com";

   // Build an array of port values
   $range =range($rangeStart, $rangeStop);

   echo "<p>Scan results for $target</p>";

   // Execute the scan
   foreach ($range as $port) {
      $result = @fsockopen($target, $port,$errno,$errstr,1);
      if ($result) echo "<p>Socket open at port $port</p>";
   }

?>
```

Scanning the `www.example.com` Web site, the following output is returned:

```
Scan results for www.example.com:
Socket open at port 22
Socket open at port 80
Socket open at port 443
```

A far lazier means for accomplishing the same task involves using a program execution command like `system()` and the wonderful free software package Nmap (`http://www.insecure.org/nmap/`). This method is demonstrated in this chapter's concluding section, "Common Networking Tasks."

pfsockopen()

```
int pfsockopen (string host, int port [, int errno [, string errstring
                [, int timeout]]])
```

The `pfsockopen()` function, or "persistent `fsockopen()`," is operationally identical to `fsockopen()`, except that the connection is not closed once the script completes execution.

Mail

This powerful yet easy-to-implement feature of PHP is so darned useful, and needed in so many Web applications, that this section is likely to be one of the more popular sections of this chapter, if not this book. In this section, you'll learn how to send e-mail using PHP's popular `mail()` function, including how to control headers, include attachments, and carry out other commonly desired tasks. Additionally, PHP's IMAP extension is introduced, accompanied by demonstrations of the numerous features made available via this great library.

This section introduces the relevant configuration directives, describes PHP's `mail()` function, and concludes with several examples highlighting this function's many usage variations.

Configuration Directives

There are five configuration directives pertinent to PHP's `mail()` function. Pay close attention to the descriptions, because each is platform-specific.

SMTP

Scope: `PHP_INI_ALL`; Default value: `localhost`

The `SMTP` directive sets the Mail Transfer Agent (MTA) for PHP's Windows platform version of the mail function. Note that this is only relevant to the Windows platform, because Unix platform implementations of this function are actually just wrappers around that operating system's mail function. Instead, the Windows implementation depends on a socket connection made to either a local or a remote MTA, defined by this directive.

sendmail_from

Scope: PHP_INI_ALL; Default value: Null

The sendmail_from directive sets the From field of the message header. This parameter is only useful on the Windows platform. If you're using a Unix platform, you must set this field within the mail function's addl_headers parameter.

sendmail_path

Scope: PHP_INI_SYSTEM; Default value: The default sendmail path

The sendmail_path directive sets the path to the sendmail binary if it's not in the system path, or if you'd like to pass additional arguments to the binary. By default, this is set to the following:

```
sendmail -t -i
```

Keep in mind that this directive only applies to the Unix platform. Windows depends upon establishing a socket connection to an SMTP server specified by the SMTP directive on port smtp_port.

smtp_port

Scope: PHP_INI_ALL; Default value: 25

The smtp_port directive sets the port used to connect to the server specified by the SMTP directive.

mail.force_extra_parameters

Scope: PHP_INI_SYSTEM; Default value: Null

You can use the mail.force_extra_parameters directive to pass additional flags to the sendmail binary. Note that any parameters passed here will replace those passed in via the mail() function's addl_parameters parameter.

As of PHP 4.2.3, the addl_params parameter is disabled if you're running in safe mode. However, any flags passed in via this directive will still be passed in even if safe mode is enabled. In addition, this parameter is irrelevant on the Windows platform.

mail()

```
boolean mail(string to, string subject, string message [, string addl_headers
             [, string addl_params]])
```

The mail() function can send an e-mail with a subject of subject and a message containing message to one or several recipients denoted in to. You can tailor many of the e-mail properties using the addl_headers parameter, and can even modify your SMTP server's behavior by passing extra flags via the addl_params parameter.

On the Unix platform, PHP's mail() function is dependent upon the sendmail MTA. If you're using an alternative MTA (qmail, for example), you need to use that MTA's sendmail wrappers. PHP's Windows implementation of the function instead depends upon establishing a socket connection to an MTA designated by the SMTP configuration directive, introduced earlier in this chapter.

The remainder of this section is devoted to numerous examples highlighting the many capabilities of this simple yet powerful function.

Sending a Plain-Text E-Mail

Sending the simplest of e-mails is trivial using the `mail()` function, done using just the three required parameters. Here's an example:

```
<?php
   mail("test@example.com", "This is a subject", "This is the mail body");
?>
```

Try swapping out the placeholder recipient address with your own and executing this on your server. The mail should arrive in your inbox within a few moments. If you've executed this script on a Windows server, the From field should denote whatever e-mail address you assigned to the `sendmail_from` configuration directive. However, if you've executed this script on a Unix machine, you might have noticed a rather odd From address, likely specifying the user `nobody` or `www`. Because of the way PHP's mail function is implemented on Unix systems, the default sender will appear as the same user under which the server daemon process is operating. You can change this default, as is demonstrated in the next example.

Sending an E-Mail with Additional Headers

The previous example was a proof-of-concept of sorts, offered just to show you that sending e-mail via PHP can indeed be done. However, it's unlikely that such a bare-bones approach would be taken in any practical implementation. Rather, you'll likely want to specify additional headers such as `Reply-To`, `Content-Type`, and `From`. To do so, you can use the `addl_headers` parameter of the `mail()` function, like so:

```
<?php
    $headers = "From:sender@example.com\r\n";
    $headers .= "Reply-To:sender@example.com\r\n";
    $headers .= "Content-Type: text/plain;\r\n charset=iso-8859-1\r\n";

    mail("test@example.com", "This is the subject",
         "This is the mail body", $headers);
?>
```

When you're using additional headers, make sure that the syntax and ordering corresponds exactly with that found in RFCs 822 and 2822; otherwise, unexpected behavior may occur. Certain mail servers have been known to not follow the specifications properly, causing additional odd behavior. Check the appropriate documentation if something appears to be awry.

Sending an E-Mail to Multiple Recipients

Sending an e-mail to multiple recipients is easily accomplished by placing the comma-separated list of addresses within the `to` parameter, like so:

```
<?php
   $headers = "From:sender@example.com\r\n";
   $recipients = "test@example.com,info@example.com";
   mail($recipients, "This is the subject","This is the mail body", $headers);
?>
```

You can also send to cc: and bcc: recipients, by modifying the corresponding headers. An example follows:

```
<?php
   $headers = "From:secretary@example.com\r\n";
   $headers .= "Bcc:theboss@example.com\n";
   mail("intern@example.com", "Company picnic scheduled",
        "Don't be late!", $headers);
?>
```

Sending an HTML-Formatted E-Mail

Although many consider HTML-formatted e-mail to rank among the Internet's greatest annoyances, nonetheless, how to send HTML-formatted e-mail is a question that comes up repeatedly in regard to PHP's `mail()` function. Therefore, it seems prudent to offer an example, and hope that no innocent recipients are harmed as a result.

Despite the widespread confusion surrounding this task, sending an HTML-formatted e-mail is actually quite easy. It's done simply by setting the `Content-Type` header to `text/html;`. Consider an example:

```
<?php

   // Assign a few headers
   $headers = "From:sender@example.com\r\n";
   $headers .= "Reply-To:sender@example.com\r\n";
   $headers .= "Content-Type: text/html;\r\n charset=\"iso-8859-1\"\r\n";

   // Create the message body.
   $body = "
   <html>
      <head>
         <title>Your Winter Quarter Schedule</title>
      </head>
      <body>
      <p>Your Winter quarter class schedule follows.<br />
      Please contact your guidance counselor should you have any questions.
      </p>
      <table>
      <tr>
         <th>Class</th><th>Teacher</th><th>Days</th><th>Time</th>
      </tr>
      <tr>
```

```
        <td>Math 630</td><td>Kelly, George</td><td>MWF</td><td>10:30am</td>
      </tr>
      <tr>
        <td>Physics 133</td><td>Josey, John</td><td>TR</td><td>1:00pm</td>
      </tr>
      </table>
      </body>
   </html>
   ";

   // Send the message
   mail("student@example.com", "Wi/03 Class Schedule", $body, $headers);

?>
```

Executing this script results in an e-mail that looks like that shown in Figure 16-1.

Wi/03 Class Schedule
sender@example.com
To: student@example.com

Your Winter quarter class schedule follows.
Please contact your guidance counselor should you have any questions.

Class	Teacher	Days	Time
Math 630	Kelly, George	MWF	10:30am
Physics 133	Josey, John	TR	1:00pm

Figure 16-1. *An HTML-formatted e-mail*

Because of the differences in the way HTML-formatted e-mail is handled by the myriad of mail clients out there, consider sticking with plain-text formatting for such matters.

Sending an Attachment

The question of how to include an attachment with a programmatically created e-mail often comes up. One of the most eloquent solutions is available via a wonderful class written and maintained by Richard Heyes (`http://www.phpguru.org/`) called HTML Mime Mail 5. Available for free download and use under the GNU GPL, it makes sending MIME-based e-mail a snap. In addition to offering the always intuitive OOP syntax for managing e-mail submissions, it's capable of executing all of the e-mail–specific tasks discussed thus far, in addition to sending attachments.

■ **Note** If the GPL license isn't suitable to your project, Richard Heyes also offers a previous release of HTML Mime Mail under the BSD license. Visit his site at `http://www.phpguru.org/` for more information.

Like most other classes, using HTML Mime Mail is as simple as placing it within your INCLUDE path, and including it into your script like so:

```
include("mimemail/htmlMimeMail5.php");
```

Next, instantiate the class and send an e-mail with a Word document included as an attachment:

```
// Instantiate the class
$mail = new htmlMimeMail5();

// Set the From and Reply-To headers
$mail->setFrom('Jason <author@example.com>');
$mail->setReturnPath('author@example.com');

// Set the Subject
$mail->setSubject('Test with attached email');

// Set the body
$mail->setText("Please find attached Chapter 16. Thank you!");

// Retrieve a file for attachment
$attachment = $mail->getFile('chapter16.doc');

// Attach the file, assigning it a name and a corresponding Mime-type.
$mail->addAttachment($attachment, 'chapter16.doc', 'application/vnd.ms-word');

// Send the email to editor@example.com
$result = $mail->send(array('editor@example.com'));
```

Keep in mind that this is only a fraction of the features offered by this excellent class. This is definitely one to keep in mind if you plan on incorporating mail-based capabilities into your application.

IMAP, POP3, and NNTP

PHP offers a powerful range of functions for communicating with the IMAP protocol, dubbed its IMAP extension. Because it's primarily used for mail, it seems fitting to place it in the "Mail" section. However, the foundational library that this extension depends upon is also capable of interacting with the POP3 and NNTP protocols. For the purposes of this introduction, this section focuses largely on IMAP-specific examples, although in many cases they will work transparently with the other two protocols.

Before delving into the specifics of the IMAP extension, however, let's take a moment to review IMAP's purpose and advantages. IMAP, an acronym for Internet Message Access Protocol, is the product of Stanford University, first appearing in 1986. However, it was at the University of Washington that the protocol really started taking hold as a popular means for accessing and manipulating remote message stores. IMAP affords the user the opportunity to manage mail as if it were local, creating and administering folders used for organization, marking mail with

various flags (read, deleted, and replied to, for example), and executing search operations on the store, among many other tasks. These features have grown increasingly useful as users require access to e-mail from multiple locations—home, office, and while traveling, for example. These days, IMAP is used just about everywhere; in fact, your own place of employment or university likely offers IMAP-based e-mail access; if not, they're way behind the technology curve.

PHP's IMAP capabilities are considerable, with almost 70 functions available through the library. This section introduces several of the key functions, and provides a few examples that, put together, offer the functionality of a very basic Web-based e-mail client. The goal of this section is to demonstrate some of the basic features of this extension and offer you a foundation upon which you can begin additional experimentation. First, however, you need to complete a few required configuration-related tasks.

Tip SquirrelMail (`http://www.squirrelmail.org/`) is a comprehensive Web-based e-mail client written using PHP and the IMAP extension. With support for 40 languages, a very active development and user community, and over 200 plug-ins, SquirrelMail remains one of the most promising open-source Web-mail products available.

Requirements

Before you can use PHP's IMAP extension, you need to complete a few relatively simple tasks, outlined in this section. PHP's IMAP extension depends on the c-client library, created and maintained by the University of Washington (UW). You can download the software from UW's FTP site, located at `ftp://ftp.cac.washington.edu/imap/`. Installing the software is trivial, and the README file located within the c-client package has instructions. However, there have been a few ongoing points of confusion, some of which are outlined here:

- The makefile contains a list of ports for many operating systems. You should choose the port that best suits your system and specify it when building the package.
- By default, the c-client software expects that you'll be performing SSL connections to the IMAP server. If you choose not to use SSL to make the connections, be sure to pass `SSLTYPE=none` along on the command line when building the package. Otherwise, PHP will fail during the subsequent configuration.
- If you plan to use the c-client library solely to allow PHP to communicate with a remote or preexisting local IMAP/POP3/NNTP server, you do not have to install the various daemons discussed in the README document. Just building the package is sufficient.
- There are reports of serious system conflicts occurring when copying the c-client source files to the operating system's `include` directory. To circumvent such problems, create a directory within that directory, called `imap-version#` for example, and place the files there.

Once the c-client build is complete, rebuild PHP using the `--with-imap` flag. To save time, review the output of the `phpinfo()` function, and copy the contents of the "Configure Command" section. This contains the last-used configure command, along with all accompanying flags. Copy that to the command line and tack the following onto it:

```
--with-imap=/path/to/c-client/directory
```

Restart Apache, and you should be ready to move on.

The following section concentrates on those functions in the library that you're most likely to use. For the sake of practicality, these functions are introduced according to their task, starting with the very basic processes, such as establishing a server connection, and ending with some of the more complicated actions you might require, such as renaming mailboxes and moving messages. Keep in mind that these are just a sampling of the functions that are made available by the IMAP extension. Consult the PHP manual for a complete listing.

Establishing and Closing a Connection

Before you do anything with one of the protocols, you need to establish a server connection. As always, once you've completed the necessary tasks, you should close the connection. This section introduces the functions that are capable of handling both tasks.

imap_open()

```
resource imap_open(string mailbox, string username, string pswd [, int options])
```

The `imap_open()` function establishes a connection to an IMAP mailbox specified by `mailbox`, returning an IMAP stream on success and `FALSE` otherwise. This connection is dependent upon three components: the `mailbox`, `username`, and `pswd`. While the latter two components are self-explanatory, it might not be so obvious that `mailbox` should consist of both the server address and the mailbox path. In addition, if the port number used isn't standard (143, 110, and 119 for IMAP, POP3, and NNTP, respectively), you need to postfix this parameter with a colon, followed by the specific port number.

The optional `options` parameter is a bitmask consisting of one or more values. The most relevant are introduced here:

- `OP_ANONYMOUS`: This NNTP-specific option should be used when you don't want to update or use the `.newsrc` configuration file.
- `CL_EXPUNGE`: This option causes the opened mailbox to be expunged upon closure. Expunging a mailbox means that all messages marked for deletion are destroyed.
- `OP_HALFOPEN`: Specifying this option tells `imap_open()` to open a connection, but not any specific mailbox. This option applies only to NNTP and IMAP.
- `OP_READONLY`: This option tells `imap_open()` to open the mailbox using read-only privileges.
- `OP_SECURE`: This option forces `imap_open()` to disregard nonsecure attempts to authenticate.

The following example demonstrates how to open connections to IMAP, POP3, and NNTP mailboxes, respectively:

```
// Open an IMAP connection
$ms = imap_open("{imap.example.com:143/imap/notls}","jason","mypswd");
```

```
// Open a POP3 connection
$ms = imap_open("{pop3.example.com:110/pop3/notls}","jason","mypswd");

// Open an NNTP connection
$ns = imap_open("{nntp.example.com:119/nntp}","jason","mypswd");
```

■ **Note** If you plan to perform a non-SSL connection, you need to postfix `mailbox` with the string `/imap/notls` for IMAP and `/pop3/notls` for POP3, because PHP assumes by default that you are using an SSL connection. Neglecting to use the postfix will cause the attempt to fail.

---

imap_close()

```
boolean imap_close(resource msg_stream [, int flag])
```

The `imap_close()` function closes a previously established stream, specified by `msg_stream`. It accepts one optional `flag`, `CL_EXPUNGE`, which destroys all messages marked for deletion upon execution. An example follows:

```
<?php

   // Open an IMAP connection
   $ms = imap_open("{imap.example.com:143}","jason","mypswd");

   // Perform some tasks ...

   // Close the connection, expunging the mailbox
   imap_close($ms, CL_EXPUNGE);

?>
```

Learning More About Mailboxes and Mail

Once you've established a connection, you can begin working with it. Some of the most basic tasks involve retrieving more information about the mailboxes and messages made available via that connection. This section introduces several of the functions that are capable of performing such tasks.

imap_getmailboxes()

```
array imap_getmailboxes(resource msg_stream, string ref, string pattern)
```

The `imap_getmailboxes()` function returns an array of objects consisting of information about each mailbox found via the stream specified by `msg_stream`. Object attributes include `name`, which denotes the mailbox name, `delimiter`, which denotes the separator between folders, and `attributes`, which is a bitmask denoting the following:

- LATT_NOINFERIORS: This mailbox has no children.
- LATT_NOSELECT: This is a container, not a mailbox.
- LATT_MARKED: This mailbox is "marked," a feature specific to the University of Washington IMAP implementation.
- LATT_UNMARKED: This mailbox is "unmarked," a feature specific to the University of Washington IMAP implementation.

The ref parameter repeats the value of the mailbox parameter used in the imap_open() function. The pattern parameter offers a means for designating the location and scope of the attempt. Setting the pattern to * returns all mailboxes, while setting it to % returns only the current level. For example, you could set pattern to /work/% to retrieve only the mailboxes found in the work directory.

Consider an example:

```
<?php
   // Designate the mail server
   $mailserver = "{imap.example.com:143/imap/notls}";

   // Establish a connection
   $ms = imap_open($mailserver,"jason","mypswd");

   // Retrieve a single-level mailbox listing
   $mbxs = imap_getmailboxes($ms, $mailserver, "INBOX/Staff/%");
   while (list($key,$val) = each($mbxs))
   {
      echo $val->name."<br />";
   }

   imap_close($ms);
?>
```

This returns:

```
{imap.example.com:143/imap/notls}INBOX/Staff/CEO
{imap.example.com:143/imap/notls}INBOX/Staff/IT
{imap.example.com:143/imap/notls}INBOX/Staff/Secretary
```

imap_num_msg()

```
int imap_num_msg(resource msg_stream)
```

This function returns the number of messages found in the mailbox specified by msg_stream. An example follows:

```
<?php
```

```
    // Open an IMAP connection
    $user = "jason";
    $pswd = "mypswd";
    $ms = imap_open("{imap.example.com:143}INBOX",$user, $pswd);

    // How many messages in user jason's inbox?
    $msgnum = imap_num_msg($ms);
    echo "<p>User $user has $msgnum messages in his inbox.</p>";

?>
```

This returns:

```
User jason has 11,386 messages in his inbox.
```

It's apparent that Jason has a serious problem organizing his messages.

■ **Tip** If you're interested in receiving just the recently arrived messages (messages that have not been included in prior sessions), check out the `imap_num_recent()` function.

imap_status()

```
object imap_status(resource msg_stream, string mbox, int options)
```

The `imap_status()` function returns an object consisting of status information pertinent to the mailbox named in `mbox`. Four possible attributes can be set, depending upon how the `options` parameter is defined. The `options` parameter can be set to one of the following values:

- `SA_ALL`: Set all of the available flags.
- `SA_MESSAGES`: Set the `messages` attribute to the number of messages found in the mailbox.
- `SA_RECENT`: Set the `recent` attribute to the number of messages recently added to the mailbox. A *recent* message is one that has not appeared in prior sessions. Note that this differs from *unseen* (unread) messages insofar as unread messages can remain unread across sessions, whereas recent messages are only deemed as such during the first session in which they appear.
- * `SA_UIDNEXT`: Set the `uidnext` attribute to the next UID used in the mailbox.
- `SA_UIDVALIDITY`: Set the `uidvalidity` attribute to a constant that changes if the UIDs for a particular mailbox are no longer valid. UIDs can be invalidated when the mail server experiences a condition that makes it impossible to maintain permanent UIDs, or when a mailbox has been deleted and re-created.
- `SA_UNSEEN`: Set the `unseen` attribute to the number of unread messages in the mailbox.

Consider the following example:

```
<?php

   $mailserver = "{mail.example.com:143/imap/notls}";
   $ms = imap_open($mailserver,"jason","mypswd");

   // Retrieve all of the attributes
   $status = imap_status($ms, $mailserver."INBOX",SA_ALL);

   // How many unseen messages?
   echo $status->unseen;
   imap_close($ms);

?>
```

This returns:

```
64
```

The majority of which are spam, no doubt!

Retrieving Messages

Obviously, you are most interested in the information found within the messages sent to you. This section shows you how to parse these messages for both header and body information.

imap_headers()

```
array imap_headers(resource msg_stream)
```

The `imap_headers()` function retrieves an array consisting of messages located in the mailbox specified by `msg_stream`. Here's an example:

```
<?php

   // Designate a mailbox and establish a connection
   $mailserver = "{mail.example.com:143/imap/notls}INBOX/Staff/CEO";
   $ms = imap_open($mailserver,"jason","mypswd");

   // Retrieve message headers
   $headers = imap_headers($ms);

   // Display total number of messages in mailbox
   echo "<strong>".count($headers)." messages in the mailbox</strong><br />";
?>
```

This returns:

```
3 messages in the mailbox
```

By itself, `imap_headers()` isn't very useful. After all, you can retrieve the total number of messages using the `imap_num_msg()` function. Instead, you typically use this function in conjunction with another function capable of parsing each of the retrieved array entries. This is demonstrated next, using the `imap_headerinfo()` function.

imap_headerinfo()

```
object imap_headerinfo(resource msg_stream, int msg_number [, int fromlength
                       [, int subjectlength [, string defaulthost]]])
```

The function `imap_headerinfo()` retrieves a vast amount of information pertinent to the message `msg_number` located in the mailbox specified by `msg_stream`. Three optional parameters can also be supplied: `fromlength`, which denotes the maximum number of characters that should be retrieved for the `from` attribute, `subjectlength`, which denotes the maximum number of characters that should be retrieved for the `subject` attribute, and `defaulthost`, which is presently a placeholder that has no purpose.

In total, 29 object attributes for each message are returned:

- `Answered`: Has the message been answered? The attribute is `A` if answered, blank otherwise.
- `bccaddress`: A string consisting of all information found in the `Bcc` header, to a maximum of 1,024 characters.
- `bcc[]`: An array of objects consisting of items pertinent to the message `Bcc` header. Each object consists of the following attributes:
  - `adl`: Known as the at-domain or source route, this attribute is deprecated and rarely, if ever, used.
  - `host`: Specifies the host component of the e-mail address. For example, if the address is `gilmore@example.com`, `host` would be set to `example.com`.
  - `mailbox`: Specifies the username component of the e-mail address. For example, if the address is `ceo@example.com`, the `mailbox` attribute would be set to `ceo`.
  - `personal`: Specifies the "friendly name" of the e-mail address. For example, the `From` header might read Jason Gilmore <gilmore@example.com>. In this case, the `personal` attribute would be set to `Jason Gilmore`.
- `ccaddress`: A string consisting of all information found in the `Cc` header, to a maximum of 1,024 characters.
- `cc[]`: An array of objects consisting of items pertinent to the message `Cc` header. Each object consists of the same attributes introduced in the `bcc[]` summary.

- `date`: The date found in the headers of the sender's mail client. Note that this can easily be incorrect or altogether forged. You'll probably want to rely on `udate` for a more accurate timeframe of when the message was received.
- `deleted`: Has the message been marked for deletion? This attribute is `D` if deleted, blank otherwise.
- `draft`: Is this message in draft format? This attribute is `X` if draft, blank otherwise.
- `fetchfrom`: The `From` header, not to exceed `fromlength` characters.
- `fetchsubject`: The `Subject` header, not to exceed `subjectlength` characters.
- `followup_to`: This attribute is used to prevent the sender's message from being sent to the user when the message is intended for a list. Note that this attribute is not standard, and is not supported by all mail agents.
- `flagged`: Has this message been flagged? This attribute is `F` if flagged, blank otherwise.
- `fromaddress`: A string consisting of all information found in the `From` header, to a maximum of 1,024 characters.
- `from[]`: An array of objects consisting of items pertinent to the message `From` header. Each object consists of the same attributes introduced in the `bcc[]` summary.
- `in_reply_to`: If the message identified by `msg_number` is in response to another message, this attribute specifies the `Message-ID` header identifying that original message.
- `message_id`: A string used to uniquely identify the message. The following is a sample message identifier:

  ```
  <1COCCEE45BOOE74D8FBBB1AE6A472E85O12C696E>@wjgilmore.com
  ```

- `newsgroups`: The newsgroups to which the message has been sent.
- `recent`: Is this message recent? This attribute is `R` if the message is recent and seen, `N` if recent and not seen, and blank otherwise.
- `reply_toaddress`: A string consisting of all information found in the `Reply-To` header, to a maximum of 1,024 characters.
- `reply_to`: An array of objects consisting of items pertinent to the `Reply-To` header. Each object consists of the same attributes introduced in the `bcc[]` summary.
- `return_path`: A string consisting of all information found in the `Return-path` header, to a maximum of 1,024 characters.
- `return_path[]`: An array of objects consisting of items pertinent to the message `Return-path` header. Each object consists of the same attributes introduced in the `bcc[]` summary.
- `subject`: The message subject.
- `senderaddress`: A string consisting of all information found in the `Sender` header, to a maximum of 1,024 characters.

- `sender`: An array of objects consisting of items pertinent to the message `Sender` header. Each object consists of the same attributes introduced in the `bcc[]` summary.
- `toaddress`: A string consisting of all information found in the `To` header, to a maximum of 1,024 characters.
- `to[]`: An array of objects consisting of items pertinent to the message `To` header. Each object consists of the same attributes introduced in the `bcc[]` summary.
- `udate`: The date the message was received by the server, formatted in Unix time.
- `unseen`: Denotes whether the message has been read. This attribute is `U` if the message is unseen and not recent, and blank otherwise.

Consider the following example:

```
<?php

   // Designate a mailbox and establish a connection
   $mailserver = "{mail.example.com:143/imap/notls}INBOX/Staff/CEO";
   $ms = imap_open($mailserver,"jason","mypswd");

   // Retrieve message headers
   $headers = imap_headers($ms);

   // Display total number of messages in mailbox
   echo "<strong>".count($headers)." messages in the mailbox</strong><br />";

   // Loop through messages and display subject/date of each
   for($x=1;$x<=count($headers);$x++)
   {
      $header = imap_header($ms,$x);
      echo $header->Subject." (".$header->Date.")<br />";
   }

   // Close the connection
   imap_close($ms);

?>
```

This returns the output shown in Figure 16-2.

3 messages in the mailbox
FWD: Weekly Status Report (Sun, 4 Aug 2004 15:08:04 -500)
Get rich quick! (Mon, 5 Aug 2004 04:27:04 -500)
RE: Course Web site (Tues, 6 Aug 2004 11:55:04 -500)

Figure 16-2. *Retrieving message headers*

Consider another example. What if you wanted to display in bold those messages that are unread? For sake of space, this example is a revision of the previous example, but includes only the relevant components:

```
<?php
...
for($x=1;$x<=count($headers);$x++)
{
   $header = imap_header($ms,$x);
   $unseen = $header->unseen;
   $recent = $header->recent;
   if ($unseen == "U" || $recent == "N") {
        $flagStart = "<strong>";
        $flagStop = "</strong>";
   }
   echo "<tr>";
   echo "<td>".$header->fromaddress."</td>";
   echo "<td>".$flagStart.$header->Subject.$flagStop."</td>";
   echo "<td>".$header->date."</td>";
   echo "</tr>";
}
echo "</table>";
...
?>
```

Note that you had to perform a Boolean test on two attributes: `recent` and `unseen`. Because `unseen` will be set to `U` if the message is unseen and not recent, and `recent` will be set to `N` if the message is recent and not seen, we can cover our bases by examining whether either is true. If so, you have found an unread message.

imap_fetchstructure()

```
object imap_fetchstructure(resource msg_stream, int msg_number [, int options])
```

The `imap_fetchstructure()` function returns an object consisting of a variety of items pertinent to the message identified by `msg_number`. If the optional `options` flag is set to `FT_UID`, then it is assumed that the `msg_number` is a UID. There are 17 different object properties, but only those that you'll probably find particularly interesting are described here:

- `bytes`: The message size, in bytes.
- `encoding`: The value assigned to the `Content-Transfer-Encoding` header. This is an integer ranging from 0 to 5, the values corresponding to `7bit`, `8bit`, `binary`, `base64`, `quoted-printable`, and `other`, respectively.
- `ifid`: This is set to `TRUE` if a `Message-ID` header exists.
- `id`: The `Message-ID` header, if one exists.
- `lines`: The number of lines found in the message body.

- type: The value assigned to the Content-Type header. This is an integer ranging from 0 to 7, the values corresponding to Text, Multipart, Message, Application, Audio, Image, Video, and Other, respectively.

Consider an example. The following code will retrieve the number of lines and size, in bytes, of a message:

```
<?php

   // Open an IMAP connection
   $user = "jason";
   $pswd = "mypswd";
   $ms = imap_open("{imap.example.com:143}INBOX", $user, $pswd);

   // Retrieve information about message number 5.
   $message = imap_fetchstructure($ms,5);
   echo "Message lines: ".$message->lines."<br />";
   echo "Message size: ".$message->bytes." bytes<br />";

?>
```

Sample output follows:

```
Message lines: 15
Message size: 854 bytes
```

imap_fetchoverview()

```
array imap_fetchoverview(resource msg_stream, string sequence [, int options])
```

The imap_fetchoverview() function retrieves the message headers for a particular sequence of messages, returning an array of objects. If the optional options flag is set to FT_UID, then it is assumed that the msg_number is a UID. Each object in the array consists of 14 attributes:

- answered: Determines whether the message is flagged as answered
- date: The date the message was sent
- deleted: Determines whether the message is flagged for deletion
- draft: Determines whether the message is flagged as a draft
- flagged: Determines whether the message is flagged
- from: The sender
- message-id: The Message-ID header
- msgno: The message's message sequence number
- recent: Determines whether the message is flagged as recent

- `references`: This message's referring `Message-ID`
- `seen`: Determines whether the message is flagged as seen
- `size`: The message's size, in bytes
- `subject`: The message's subject
- `uid`: The message's UID

Among other things, you can use this function to produce a listing of messages that have not yet been read:

```
<?php
   // Open an IMAP connection
   $user = "jason";
   $pswd = "mypswd";
   $ms = imap_open("{imap.example.com:143}INBOX",$user, $pswd);

   // Retrieve total number of messages
   $nummsgs = imap_num_msg($ms);
   $messages = imap_fetch_overview($ms,"1:$nummsgs");

   // If message not flagged as seen, output info about it
   while(list($key,$value) = each($messages)) {
      if ($value->seen == 0) {
         echo "<p>Subject: ".$value->subject."<br />";
         echo "Date: ".$value->date."<br />";
         echo "From: ".$value->from."</p>";
      }
   }
?>
```

Sample output follows:

```
Subject: Audio Visual Web site
Date: Mon, 26 Aug 2006 18:04:37 -0500
From: Andrew Fieldpen

Subject: The Internet is broken
Date: Mon, 27 Aug 2006 20:04:37 -0500
From: "Roy J. Dugger"

Subject: Re: Standards article for Web browsers
Date: Mon, 28 Aug 2006 21:04:37 -0500
From: Nicholas Kringle
```

Note the use of a colon to separate the starting and ending message numbers. Also, keep in mind that this function will always sort the array in ascending order, even if you place the ending message number first. Finally, it's possible to selectively choose messages by separating each number with a comma. For example, if you want to retrieve information about messages 1 through 3, and 5, you can set sequence like so: 1:3,5.

imap_fetchbody()

```
string imap_fetchbody(resource msg_stream, int msg_number,
                      string part_number [, flags options])
```

The imap_fetchbody() function retrieves a particular section (part_number) of the message body identified by msg_number, returning the section as a string. The optional options flag is a bitmask containing one or more of the following items:

- FT_UID: Consider the msg_number value to be a UID.
- FT_PEEK: Do not set the message's seen flag if it isn't already set.
- FT_INTERNAL: Do not convert any newline characters. Instead, return the message exactly as it appears internally to the mail server.

If you leave part_number blank, by assigning it an empty string, this function returns the entire message text. You can selectively retrieve message parts by assigning part_number an integer value denoting the message part's position. The following example retrieves the entire message:

```
<?php

   // Open an IMAP connection
   $user = "jason";
   $pswd = "mypswd";
   $ms = imap_open("{imap.example.com:143}INBOX",$user, $pswd);

   $message = imap_fetchbody($ms,1,"","FT_PEEK");
   echo $message;

?>
```

Sample output follows:

```
Jason,

Can we create a Web administrator account for my new student?

Thanks
Bill Niceguy
```

```
From: "Josh Crabgrass" <crabgrass@example.com>
To: "'Bill Niceguy'" <niceguy@example.com>
Subject: RE: Web site access
Date: Mon, 5 August 2004 10:26:01 -0400
X-Mailer: Microsoft Outlook, Build 10.0.4510
Importance: Normal

Bill,

I'll need an admin account in order to maintain the new Web site.

Thanks,
Josh
```

Composing a Message

Creating and sending messages are likely the two e-mail tasks that take up most of your time. The next two functions demonstrate how both are accomplished using PHP's IMAP extension.

imap_mail_compose()

```
string imap_mail_compose(array envelope, array body)
```

This function creates a MIME message based on the provided `envelope` and `body`. The `envelope` comprises all of the header information pertinent to the addressing of the message, including well-known items such as `From`, `Reply-To`, `CC`, `BCC`, `Subject`, and others. The `body` consists of the actual message and various attributes pertinent to its format. Once created, you can do any number of things with the message, including mailing it, appending it to an existing mail store, or anything else for which MIME messages are suitable.

A basic composition example follows:

```
<?php

   $envelope["from"] = "gilmore@example.com";
   $envelope["to"] = "admin@example.com";
   $msgpart["type"] = TYPETEXT;
   $msgpart["subtype"] = "plain";
   $msgpart["contents.data"] = "This is the message text.";
   $msgbody[1] = $msgpart;

   echo nl2br(imap_mail_compose($envelope,$msbody));

?>
```

The following example returns:

```
From: gilmore@example.com
To: admin@example.com
MIME-Version: 1.0
Content-Type: TEXT/plain; CHARSET=US-ASCII
This is the message text.
```

Sending a Message

Once you've composed a message, you can send it using the `imap_mail()` function, introduced next.

imap_mail()

```
boolean imap_mail(string rcpt, string subject, string msg
                  [, string addl_headers [, string cc [, string bcc
                  [, string rpath]]]])
```

The `imap_mail()` function works much like the previously introduced `mail()` function, sending a message to the address specified by `rcpt`, possessing the subject of `subject` and the message consisting of `msg`. You can include additional headers with the parameter `addl_headers`. In addition, you can CC and BCC additional recipients with the parameters `cc` and `bcc`, respectively. Finally, the `rpath` parameter is used to set the `Return-path` header.

Let's revise the previous example so that the composed message is also sent:

```
<?php

   $envelope["from"] = "gilmore@example.com";
   $msgpart["type"] = TYPETEXT;
   $msgpart["subtype"] = "plain";
   $msgpart["contents.data"] = "This is the message text.";
   $msgbody[1] = $msgpart;
   $message = imap_mail_compose($envelope,$msbody);

   // Separate the message header and body. Some
   // mail clients seem unable to do so.

   list($msgheader,$msgbody)=split("\r\n\r\n",$message,2);
   $subject = "Test IMAP message";
   $to = "jason@example.com";
   $result=imap_mail($to,$subject,$msgbody,$msgheader);

?>
```

Mailbox Administration

IMAP offers the ability to organize mail by categorizing it within compartments commonly referred to as *folders* or *mailboxes*. This section shows you how to create, rename, and delete these mailboxes.

imap_createmailbox()

```
boolean imap_createmailbox(resource msg_stream, string mbox)
```

The `imap_createmailbox()` function creates a mailbox named `mbox`, returning `TRUE` on success and `FALSE` otherwise. The following example uses this function to create a mailbox residing at the user's top level (`INBOX`):

```
<?php
   $mailserver = "{imap.example.com:143/imap/notls}INBOX";
   $mbox = "events";
   $ms = imap_open($mailserver,"jason","mypswd");
   imap_createmailbox($ms,$mailserver."/".$mbox);
   imap_close($ms);
?>
```

Take note of the syntax used to specify the mailbox path:

```
{imap.example.com:143/imap/notls}INBOX/events
```

As is the case with many of PHP's IMAP functions, the entire server string must be referenced as if it were part of the mailbox name itself.

imap_deletemailbox()

```
boolean imap_deletemailbox(resource msg_stream, string mbox)
```

The `imap_deletemailbox()` function deletes an existing mailbox named `mbox`, returning `TRUE` on success and `FALSE` otherwise. For example:

```
<?php
   $mbox = "{imap.example.com:143/imap/notls}INBOX";
   if (imap_deletemailbox($ms, "$mbox/staff"))
      echo "The mailbox has successfully been deleted.";
   else
      echo "There was a problem deleting the mailbox";
?>
```

Keep in mind that deleting a mailbox also deletes all mail found in that mailbox.

imap_renamemailbox()

```
boolean imap_renamemailbox(resource msg_stream, string old_mbox, string new_mbox)
```

The imap_renamemailbox() function renames an existing mailbox named old_mbox to new_mbox, returning TRUE on success and FALSE otherwise. An example follows:

```
<?php
   $mbox = "{imap.example.com:143/imap/notls}INBOX";
   if (imap_renamemailbox($ms, "$mbox/staff", "$mbox/teammates"))
      echo "The mailbox has successfully been renamed";
   else
      echo "There was a problem renaming the mailbox";
?>
```

Message Administration

One of the beautiful aspects of IMAP is that you can manage mail from anywhere. This section offers some insight into how this is accomplished using PHP's functions.

imap_expunge()

```
boolean imap_expunge(resource msg_stream)
```

The imap_expunge() function destroys all messages flagged for deletion, returning TRUE on success and FALSE otherwise. Note that you can automate this process by including the CL_EXPUNGE flag on stream creation or closure.

imap_mail_copy()

```
boolean imap_mail_copy(resource msg_stream, string msglist, string mbox
                       [, int options])
```

The imap_mail_copy() function copies the mail messages located within msglist to the mailbox specified by mbox. The optional options parameter is a bitmask that accounts for one or more of the following flags:

- *CP_UID: The msglist consists of UIDs instead of message index identifiers.
- *CP_MOVE: Including this flag deletes the messages from their original mailbox after the copy is complete.

imap_mail_move()

```
boolean imap_mail_move(resource msg_stream, string msglist, string mbox
                       [, int options])
```

The imap_mail_move() function moves the mail messages located in msglist to the mailbox specified by mbox. The optional options parameter is a bitmask that accepts the following flag:

CP_UID: The msglist consists of UIDs instead of message index identifiers.

Streams

These days, even trivial Web applications often consist of a well-orchestrated blend of programming languages and data sources. In many such instances, interaction between the language and data source involves reading or writing a linear stream of data, known as a *stream*. For example, invoking the command `fopen()` results in the binding of a file name to a stream. At that point, that stream can be read from and written to, depending upon the invoked mode setting and upon permissions.

Although you might immediately think of calling `fopen()` on a local file, you might find it interesting to know that you can also create stream bindings using a variety of methods, over HTTP, HTTPS, FTP, FTPS, and even compress the stream using the zlib and bzip2 libraries. This is accomplished using an appropriate *wrapper,* of which PHP supports several. This section talks a bit about streams, focusing on stream wrappers and another interesting concept known as *stream filters*.

■ **Note** PHP 5 introduces an API for creating and registering your own stream wrappers and filters. An entire book could be devoted to the topic, but the matter would simply not be of interest to the majority of readers. Therefore, there is no coverage of the matter in this book. If you are interested in learning more, please consult the PHP manual.

Stream Wrappers and Contexts

A *stream wrapper* is a bit of code that wraps around the stream, managing the stream in accordance with a specific protocol, be it HTTP, FTP, or otherwise. Because PHP supports several wrappers by default, you can bind streams over these protocols transparently, like so:

```
<?php
   echo file_get_contents("http://www.example.com/");
?>
```

Executing this returns the contents of the `www.example.com` domain's index page:

```
You have reached this web page by typing "example.com", "example.net",
or "example.org" into your web browser.
These domain names are reserved for use in documentation and are not
available for registration. See RFC 2606, Section 3.
```

As you can see, no other code was involved for handling the fact that an HTTP stream binding was performed. PHP transparently supports binding for the following types of streams: HTTP, HTTPS, FTP, FTPS, file system, PHP input/output, and compression.

From Chapter 10, you may remember that the `fopen()` function accepts a parameter named `zcontext`. Now that you're a bit more familiar with streams and wrappers, this seems

like an opportune time to introduce contexts. Simply put, a *context* is a set of wrapper-specific options that tweaks a stream's behavior. Each supported stream wrapper offers its own set of options. You can reference these options in the PHP manual on your own; however, to give you an idea, this section demonstrates how one such option can modify a stream's behavior. To use any such context, you first need to create it using the `stream_context_create()` function, introduced next.

stream_context_create()

```
resource stream_context_create(array options)
```

The `stream_context_create()` function creates a resource context based on the array of options passed to it. Its purpose is best illustrated with an example. By default, FTP streams do not permit the overwriting of existing files on a remote server. Sometimes, though, you may wish to enable this behavior. To do so, you first need to create a context resource, passing in the `overwrite` parameter, and then pass that resource to set `fopen()`'s `zcontext` parameter. This process is made apparent in the following code:

```
<?php
   $params = array("ftp" => array("overwrite" => "1"));
   $context = stream_context_create($params);
   $fh = fopen("ftp://localhost/", "w", 0, $context);
?>
```

Stream Filters

Sometimes you need to manipulate stream data either as it is read in from or as it is written to some data source. For example, you might want to strip all HTML tags from a stream. Using a *stream filter,* this is a trivial matter. At the time of this writing, three types of stream filters are available: string, conversion, and compression. As of PHP version 5.0 RC1, the string and conversion types were available by default. You can enable the compression filters by installing the `zlib_filter` package, available via PECL (`http://pecl.php.net/`). Table 16-1 offers a list of default filters and their corresponding descriptions.

Table 16-1. *PHP's Default Stream Filters*

Filter	Description
`string.rot13`	See the standard PHP function `rot13()`.
`string.toupper`	See the standard PHP function `toupper()`.
`string.tolower`	See the standard PHP function `tolower()`.
`string.strip_tags`	See the standard PHP function `strip_tags()`.
`convert.base64-encode`	See the standard PHP function `base64_encode()`.
`convert.base64-decode`	See the standard PHP function `base64_decode()`.
`convert.quoted-printable-decode`	See the standard PHP function `quoted_printable_decode()`.

Table 16-1. *PHP's Default Stream Filters (Continued)*

Filter	Description
convert.quoted-printable-encode	No functional equivalent. In addition to the parameters supported by base64_encode(), it also supports the Boolean arguments binary and force-encode-first, in that order. These arguments specify, respectively, whether the stream should be handled in binary format and whether it should be first encoded using base64_encode().

To view the filters available to your PHP distribution, use the stream_get_filters() function, introduced next.

stream_get_filters()

```
array stream_get_filters()
```

The stream_get_filters() function returns an array of all registered stream filters. Consider an example:

```
print_r(stream_get_filters());
```

This example returns:

```
Array (
   [0] => convert.iconv.*
   [1] => string.rot13
   [2] => string.toupper
   [3] => string.tolower
   [4] => string.strip_tags
   [5] => convert.*
)
```

It isn't clear why this function lists all of the available string-based filters but masks the names of the two conversion filters by consolidating the group using an asterisk. As of PHP version 5.0 RC1, there are four conversion filters, namely base64-encode, base64-decode, quoted-printable-encode, and quoted-printable-decode.

To use a filter, you need to pass it through one of two functions, stream_filter_append() or stream_filter_prepend(). Which one you choose depends on the order in which you'd like to execute the filter in respect to any other assigned filters. Both functions are introduced next.

stream_filter_append()

```
boolean stream_filter_append(resource stream, string filtername
                             [,int read_write [, mixed params]])
```

The stream_filter_append() function appends the filter filtername to the end of a list of any filters currently being executed against stream. The optional read_write parameter specifies the filter chain (read or write) to which the filter should be applied. Typically you won't need this because PHP will take care of it for you, by default. The final optional parameter, params, specifies any parameters that are to be passed into the filter function.

Let's consider an example. Suppose you're writing a form-input blog entry to an HTML file. The only allowable HTML tag is
, so you'll want to remove all other characters from the stream as it's written to the HTML file:

```
<?php
$blog = <<< blog
One of my <b>favorite</b> blog tools is Movable Type.<br />
You can learn more about Movable Type at
<a href="http://www.movabletype.org/">http://www.movabletype.org/</a>.
blog;

   $fh = fopen("042006.html", "w");
   stream_filter_append($fh, "string.strip_tags", STREAM_FILTER_WRITE, "<br>");
   fwrite($fh, $blog);
   fclose($fh);
?>
```

If you open up 042006.html, you'll find the following contents:

```
One of my favorite blog tools is Movable Type.<br />
You can learn more about Movable Type at http://www.movabletype.org/.
```

stream_filter_prepend()

```
boolean stream_filter_prepend(resource stream, string filtername
                              [,int read_write [, mixed params]])
```

The function stream_filter_prepend() prepends the filter filtername to the front of a list of any filters currently being executed against stream. The optional read_write and params parameters correspond in purpose to those described in stream_filter_append().

Common Networking Tasks

Although various command-line applications have long been capable of performing the networking tasks demonstrated in this section, offering a means for carrying them out via the Web certainly can be useful. For example, at work we host a variety of such Web-based applications within our intranet for the IT support department employees to use when they are troubleshooting a networking problem but don't have an SSH client handy. In addition, they can be accessed via Web browsers found on most modern wireless PDAs. Finally, although the command-line counterparts are far more powerful and flexible, viewing such information via the Web is at times simply more convenient. Whatever the reason, it's likely you could put to good use some of the applications found in this section.

■ **Note** Several examples in this section use the `system()` function. This function is introduced in Chapter 10.

Pinging a Server

Verifying a server's connectivity is a commonplace administration task. The following example shows you how to do so using PHP:

```
<?php

   // Which server to ping?
   $server = "www.example.com";

   // Ping the server how many times?
   $count = 3;

   // Perform the task
   echo "<pre>";
   system("/bin/ping -c $count $server");
   echo "</pre>";

   // Kill the task
   system("killall -q ping");

?>
```

The preceding code should be fairly straightforward, except for perhaps the system call to `killall`. This is necessary because the command executed by the system call will continue to execute if the user ends the process prematurely. Because ending execution of the script within the browser will not actually stop the process for execution on the server, you need to do it manually.

Sample output follows:

```
PING www.example.com (192.0.34.166) from 123.456.7.8 : 56(84) bytes of data.
64 bytes from www.example.com (192.0.34.166): icmp_seq=0 ttl=255 time=158 usec
64 bytes from www.example.com (192.0.34.166): icmp_seq=1 ttl=255 time=57 usec
64 bytes from www.example.com (192.0.34.166): icmp_seq=2 ttl=255 time=58 usec

--- www.example.com ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.048/0.078/0.158/0.041 ms
```

PHP's program execution functions are great because they allow you to take advantage of any program installed on the server. We'll return to these functions several times throughout this section.

A Port Scanner

The introduction of `fsockopen()` earlier in this chapter was accompanied by a demonstration of how to create a port scanner. However, like many of the tasks introduced in this section, this can be accomplished much more easily using one of PHP's program execution functions. The following example uses PHP's `system()` function and the Nmap (network mapper) tool:

```
<?php

   $target = "www.example.com";
   echo "<pre>";
   system("/usr/bin/nmap $target");
   echo "</pre>";

   // Kill the task
   system("killall -q nmap");

?>
```

A snippet of the sample output follows:

```
Starting nmap V. 2.54BETA31 ( www.insecure.org/nmap/ )
Interesting ports on  (209.51.142.155):
(The 1500 ports scanned but not shown below are in state: closed)
Port        State       Service
22/tcp     open       ssh
80/tcp     open       http
110/tcp   open       pop-3
111/tcp   filtered     sunrpc
```

Subnet Converter

You've probably at one time scratched your head trying to figure out some obscure network configuration issue. Most commonly, the culprit for such woes seems to center on a faulty or unplugged network cable. Perhaps the second most common problem one faces is a mistake made when calculating the necessary basic network ingredients: IP addressing, subnet mask, broadcast address, network address, and the like. To remedy this, a few PHP functions and bitwise operations can be coaxed into doing the calculations for you. The example shown in Listing 16-2 calculates several of these components, given an IP address and a bitmask.

Listing 16-2. *A Subnet Converter*

```
<form action="netaddr.php" method="post">
<p>
IP Address:<br />
<input type="text" name="ip[]" size="3" maxlength="3" value="" />.
<input type="text" name="ip[]" size="3" maxlength="3" value="" />.
<input type="text" name="ip[]" size="3" maxlength="3" value="" />.
<input type="text" name="ip[]" size="3" maxlength="3" value="" />
</p>

<p>
Subnet Mask:<br />
<input type="text" name="sm[]" size="3" maxlength="3" value="" />.
<input type="text" name="sm[]" size="3" maxlength="3" value="" />.
<input type="text" name="sm[]" size="3" maxlength="3" value="" />.
<input type="text" name="sm[]" size="3" maxlength="3" value="" />
</p>

<input type="submit" name="submit" value="Calculate" />

</form>

<?php
   if (isset($_POST['submit']))
   {
      // Concatenate the IP form components and convert to IPv4 format
      $ip = implode('.',$_POST['ip']);
      $ip = ip2long($ip);

      // Concatenate the netmask form components and convert to IPv4 format
      $netmask = implode('.',$_POST['nm'];
      $netmask = ip2long($netmask);

      // Calculate the network address
      $na = ($ip & $netmask);
      // Calculate the broadcast address
      $ba = $na | (~$netmask);

      // Convert the addresses back to the dot-format representation and display
      echo "Addressing Information: <br />";
          echo "<ul>";
         echo "<li>IP Address: ". long2ip($ip)."</li>";
         echo "<li>Subnet Mask: ". long2ip($netmask)."</li>";
         echo "<li>Network Address: ". long2ip($na)."</li>";
         echo "<li>Broadcast Address: ". long2ip($ba)."</li>";
         echo "<li>Total Available Hosts: ".($ba - $na - 1)."</li>";
         echo "<li>Host Range: ". long2ip($na + 1)." -  ".
```

```
                long2ip($ba - 1)."</li>";
        echo "</ul>";
}
?>
```

Consider an example. If you supply 192.168.1.101 as the IP address and 255.255.255.0 as the subnet mask, you should see the output shown in Figure 16-3.

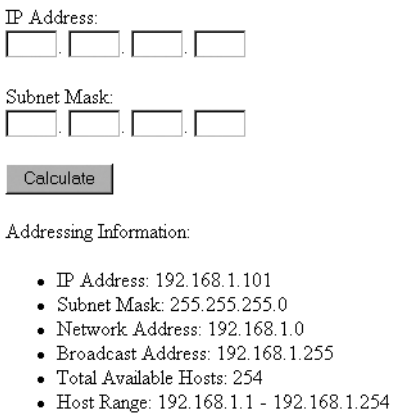

Figure 16-3. *Calculating network addressing*

Testing User Bandwidth

Although various forms of bandwidth-intensive media are commonly used on today's Web sites, keep in mind that not all users have the convenience of a high-speed network connection at their disposal. You can automatically test a user's network speed with PHP by sending the user a relatively large amount of data and then noting the time it takes for transmission to complete.

Create the data file that will be transmitted to the user. This can be anything, really, because the user will never actually see the file. Consider creating it by generating a large amount of text and writing it to a file. For example, this script will generate a text file that is roughly 1,500 KB in size:

```
<?php
   // Create a new file, creatively named "textfile.txt"
   $fh = fopen("textfile.txt","w");
   // Write the word "bandwidth" repeatedly to the file.
   for ($x=0;$x<170400;$x++) fwrite($fh,"bandwidth");
   // Close the file
   fclose($fh);
?>
```

Now we'll write the script that will calculate the network speed. This script is shown in Listing 16-3.

Listing 16-3. *Calculating Network Bandwidth*

```
<?php

   // Retrieve the data to send to the user
   $data = file_get_contents("textfile.txt");

   // Determine the data's total size, in Kilobytes
   $fsize = filesize("textfile.txt") / 1024;

   // Define the start time
   $start = time();

   // Send the data to the user
   echo "<!-- $data -->";

   // Define the stop time
   $stop = time();

   // Calculate the time taken to send the data
   $duration = $stop - $start;

   // Divide the file size by the number of seconds taken to transmit it
   $speed = round($fsize / $duration,2);

   // Display the calculated speed in Kilobytes per second
   echo "Your network speed: $speed KB/sec.";

?>
```

Executing this script produces output similar to the following:

```
Your network speed: 249.61 KB/sec.
```

Summary

PHP's networking capabilities won't soon replace those tools already offered on the command line or other well-established clients. Nonetheless, as PHP's command-line capabilities continue to gain traction, it's likely you'll quickly find a use for some of the material presented in this chapter.

The next chapter introduces one of the most powerful examples of how PHP can effectively interact with other enterprise technologies, showing you just how easy it is to interact with your preferred directory server using PHP's LDAP extension.