SOCKET PROGRAMMING with PHP

By icarus

Table of Contents

Doing A Deal

If you've been working with PHP for a while, you're probably used to thinking about it only in the context of a Web page or a Web server. While this is not unusual − PHP is, after all, a scripting language that's most commonly used to dynamically generate Web pages − it can stifle your creativity; as you'll see in this article, there's a lot more you can do with PHP than just connect to a database, retrieve records and insert them into an HTML template.

One of PHP's better−guarded secrets − and one that I discovered quite by accident − is a very comprehensive set of network programming functions. Steadily evolving over the last few releases, this socket programming API now supports almost everything you would need for socket−based client−server communication over TCP/IP, and can be rapidly deployed to build simple client−server applications in PHP.

Over the course of this article, I'll be taking a close look at the more important functions in this API, using code listings and explanations to put them in context. Along the way, I'll also build some simple client−server applications, thereby illustrating PHP's effectiveness as a rapid development and deployment tool for network−based applications through practical, real−life examples (well, some of them, at any rate).

It should be noted at the outset itself that this is an introductory article, primarily meant for Web developers interested in expanding their PHP knowledge into the somewhat−arcane area of network programming. I won't be getting into anything too complicated − for truly sophisticated client−server programming, you're probably better off with C++ or Java anyway − preferring instead to focus on the practical applications of PHP's socket functions. As is traditional with these articles, I'll also keep the technical jargon to a minimum, avoiding mention of tongue−twisting acronyms and hard−to−pronounce buzzwords.

In return for all this largesse, I expect you to laugh at the appropriate places, and say nice things about me to your friends. Do we have a deal?

Going Backwards

For those of you new to the topic, a "socket" provides a way for clients and servers to communicate in a networked environment. It creates an end−to−end communication channel, allowing a client to send requests to a server and a server to receive these requests and respond to them. A common example here is that of a Web server; the server opens a socket (usually on port 80), and clients (Web browsers) communicate with it through this socket, requesting specific HTML pages for processing and display.

PHP comes with a fairly comprehensive set of functions to create and manipulate socket communications; however, this capability is not enabled by default. Consequently, you may need to recompile your PHP binary with the "−−enable−sockets" parameter to activate socket support. Windows users get a pre−built binary with their distribution.

Let's start with a simple example − a TCP server that accepts a string as input, reverses it and returns it to the client. Here's the code:

```
<?

// set some variables
$host = "192.168.1.99";
$port = 1234;

// don't timeout!
set_time_limit(0);

// create socket
$socket = socket_create(AF_INET, SOCK_STREAM, 0) or die("Could
not create
socket\n");

// bind socket to port
$result = socket_bind($socket, $host, $port) or die("Could not
bind to
socket\n");

// start listening for connections
$result = socket_listen($socket, 3) or die("Could not set up
socket
listener\n");

// accept incoming connections
// spawn another socket to handle communication
$spawn = socket_accept($socket) or die("Could not accept
incoming
connection\n");

// read client input
```

```
$input = socket_read($spawn, 1024) or die("Could not read
input\n");

// clean up input string
$input = trim($input);

// reverse client input and send back
$output = strrev($input) . "\n";
socket_write($spawn, $output, strlen ($output)) or die("Could
not write
output\n");

// close sockets
socket_close($spawn);
socket_close($socket);
?>
```

This is somewhat involved, so an explanation is in order:

1. The first step here is to set up a couple of variables to hold information on the IP address and port on which the socket server will run. You can set up your server to use any port in the numeric range 1−65535, so long as that port is not already in use.

```
<?
// set some variables
$host = "192.168.1.99";
$port = 1234;
?>
```

2. Since this is a server, it's also a good idea to use the set_time_limit() function to ensure that PHP doesn't time out and die() while waiting for incoming client connections.

```
<?
// don't timeout!
set_time_limit(0);
?>
```

3. With the preliminaries out of the way, it's time to create a socket with the socket_create() function − this function returns a socket handle that must be used in all subsequent function calls.

```
<?
// create socket
$socket = socket_create(AF_INET, SOCK_STREAM, 0) or die("Could
```

```
not create
socket\n");
?>
```

In case you're wondering what this is all about, don't worry too much about it. The AF_INET parameter specifies the domain, while the SOCK_STREAM parameter tells the function what type of socket to create (in this case, TCP).

If you wanted to create a UDP socket, you could use the following line of code instead:

```
<?
// create socket
$socket = socket_create(AF_INET, SOCK_DGRAM, 0) or die("Could
not create
socket\n");
?>
```

4. Once a socket handle has been created, the next step is to attach, or "bind", it to the specified address and port. This is accomplished via the socket_bind() function.

```
<?
// bind socket to port
$result = socket_bind($socket, $host, $port) or die("Could not
bind to
socket\n");
?>
```

5. With the socket created and bound to a port, it's time to start listening for incoming connections. PHP allows you to set the socket up as a listener via its socket_listen() function, which also allows you to specify the number of queued connections to allow as a second parameter (3, in this example).

```
<?
// start listening for connections
$result = socket_listen($socket, 3) or die("Could not set up
socket
listener\n");
?>
```

6. At this point, your server is basically doing nothing, waiting for incoming client connections. Once a client connection is received, the socket_accept() function springs into action, accepting the connection request and spawning another child socket to handle messaging between the client and the server.

```
<?
// accept incoming connections
// spawn another socket to handle communication
$spawn = socket_accept($socket) or die("Could not accept
incoming
connection\n");
?>
```

This child socket will now be used for all subsequent communication between the client and server.

7. With a connection established, the server now waits for the client to send it some input − this input is read via the socket_read() function, and assigned to the PHP variable $input.

```
<?
// read client input
$input = socket_read($spawn, 1024) or die("Could not read
input\n");
?>
```

The second parameter to socket_read() specifies the number of bytes of input to read − you can use this to limit the size of the data stream read from the client.

Note that the socket_read() function continues to read data from the client until it encounters a carriage return (\n), a tab (\t) or a \0 character. This character as treated as the end−of−input character, and triggers the next line of the PHP script.

8. The server now must now process the data sent by the client − in this example, this processing merely involves reversing the input string and sending it back to the client. This is accomplished via the socket_write() function, which makes it possible to send a data stream back to the client via the communication socket.

```
<?
// reverse client input and send back
$output = strrev($input) . "\n";
socket_write($spawn, $output, strlen ($output)) or die("Could
not write
output\n");
?>
```

The socket_write() function needs three parameters: a reference to the socket, the string to be written to it, and the number of bytes to be written.

9. Once the output has been sent back to the client, both generated sockets are terminated via the

socket_close() function.

```
<?
// close sockets
socket_close($spawn);
socket_close($socket);
?>
```

And that's it − socket creation, in nine easy steps!

Putting It All Together

Now, how about seeing it in action? Since this script generates an "always−on" socket, it isn't a good idea to run it via your Web server; instead, you might prefer to run it from the command line via the PHP binary:

```
$ /usr/local/bin/php −q server.php
```

In case you don't have a PHP binary, it's fairly easy to compile one − just follow the installation instructions for compiling a static Apache module (these instructions are available in the PHP distribution), but omit the "−−with−apache" parameter to the "configure" script.

Note the additional −q parameter to PHP − this tells the program to suppress the "Content−Type: text/html" header that it usually adds when executing a script (I don't need this header here because the output of this script isn't going to a browser).

Once the script has been executed and the socket server is active, you can simply telnet to it using any standard telnet application, and send it a string of characters as input. The server should respond with the reversed string, and then terminate the connection. Here's what it looks like:

```
$ telnet 192.168.1.99 1234
Trying 192.168.1.99...
Connected to medusa.
Escape character is '^]'.
jack and the beanstalk
klatsnaeb eht dna kcaj
Connection closed by foreign host.
```

Fortune's Fool

The steps demonstrated on the previous page make up a fairly standard process flow for constructing a socket server in PHP, and almost every server you create will follow the same basic steps. Consider the following example, which modifies the previous example to produce a random message from the "fortune" program every time a client connects to the server:

```
<?

// don't timeout!
set_time_limit(0);

// set some variables
$host = "192.168.1.99";
$port = 1234;
$command = "/usr/games/fortune";

// create socket
$socket = socket_create(AF_INET, SOCK_STREAM, 0) or die("Could
not create
socket\n");

// bind socket to port
$result = socket_bind($socket, $host, $port) or die("Could not
bind to
socket\n");

// start listening for connections
$result = socket_listen($socket, 3) or die("Could not set up
socket
listener\n");

echo "Waiting for connections...\n";

// accept incoming connections
// spawn another socket to handle communication
$spawn = socket_accept($socket) or die("Could not accept
incoming
connection\n");

echo "Received connection request\n";

// run command and send back output
$output = `$command`;
socket_write($spawn, $output, strlen ($output)) or die("Could
not write
output\n");
```

```
echo "Sent output: $output\n";

// close sockets
socket_close($spawn);
socket_close($socket);
echo "Socket terminated\n";
?>
```

In this case, I'm not even waiting to receive any data from the client. Instead, I'm simply executing an external command (the "fortune" program) with PHP's backtick(`) operator, sending the results of the command to the client, and closing the connection. A trifle abrupt, but you know how rude young people are nowadays.

Note the addition of debug messages on the server side of the connection − these messages provide a handy way to find out the current status of the socket.

Here's an example of what a client sees when it connects to the socket,

```
$ telnet 192.168.1.99 1234
Trying 192.168.1.99...
Connected to medusa.
Escape character is '^]'.
Paradise is exactly like where you are right now ... only
much, much better.
-- Laurie Anderson
Connection closed by foreign host.
```

and here are the corresponding debug messages generated on the server:

```
$ /usr/local/bin/php -q server.php
Waiting for connections...
Received connection request
Sent output: Paradise is exactly like where you are right now
... only
much, much better.
-- Laurie Anderson

Socket terminated
```

Looping The Loop

You can also write a server that allows for more than just one transaction at a time. Consider the following variant of the first example:

```
<?

// don't timeout
set_time_limit (0);

// set some variables
$host = "192.168.1.99";
$port = 1234;

// create socket
$socket = socket_create(AF_INET, SOCK_STREAM, 0) or die("Could
not create
socket\n");

// bind socket to port
$result = socket_bind($socket, $host, $port) or die("Could not
bind to
socket\n");

// start listening for connections
$result = socket_listen($socket, 3) or die("Could not set up
socket
listener\n");

echo "Waiting for connections...\n";

// accept incoming connections
// spawn another socket to handle communication
$spawn = socket_accept($socket) or die("Could not accept
incoming
connection\n");

echo "Received connection request\n";

// write a welcome message to the client
$welcome = "Roll up, roll up, to the greatest show on
earth!\n? ";
socket_write($spawn, $welcome, strlen ($welcome)) or
die("Could not send
connect string\n");

// keep looping and looking for client input
```

```
do
{
// read client input
$input = socket_read($spawn, 1024, 1) or die("Could not read
input\n");

if (trim($input) != "")
{
echo "Received input: $input\n";

// if client requests session end
if (trim($input) == "END")
{
// close the child socket
// break out of loop
socket_close($spawn);
break;
}
// otherwise...
else
{
// reverse client input and send back
$output = strrev($input) . "\n";
socket_write($spawn, $output . "? ", strlen (($output)+2)) or
die("Could
not write output\n");
echo "Sent output: " . trim($output) . "\n";
}
}

} while (true);

// close primary socket
socket_close($socket);
echo "Socket terminated\n";
?>
```

This is almost exactly the same code as that used in the very first example, with the obvious addition of debug messages and a "do−while" loop which allows the server to keep reading new data transmitted by the client, and responding to it. The loop is terminated when the client sends the special session terminator string "END".

Here's the output of a sample session:

```
$ telnet 192.168.1.99 1234
Trying 192.168.1.99...
Connected to medusa.
```

```
Escape character is '^]'.
Roll up, roll up, to the greatest show on earth!
? Look Ma...all backwards
sdrawkcab lla...aM kooL
? jack frost
tsorf kcaj
? END
Connection closed by foreign host.
```

And here are the corresponding debug messages generated on the server:

```
$ /usr/local/bin/php -q server.php
Waiting for connections...
Received connection request
Received input: Look Ma...all backwards
Sent output: sdrawkcab lla...aM kooL
Received input: jack frost
Sent output: tsorf kcaj
Received input: END
Socket terminated
```

On Web−bed Feet

Thus far, I've been using a standard telnet client to connect to my socket server and interact with it. However, it's just as easy to write a simple socket client in PHP. Consider the following example, which requests user input through an HTML form and creates a client connection to the server demonstrated a few pages back. The user's input is sent from the client to the server via this newly−minted socket connection, and the return value from the server (the same string, but reversed) is displayed to the user on an HTML page.

```
<html>
<head>
</head>

<body>

<?
// form not yet submitted
if (!$submit)
{
?>
<form action="<? echo $PHP_SELF; ?>" method="post">
Enter some text:<br>
<input type="Text" name="message" size="15"><input
type="submit"
name="submit" value="Send">
</form>
<?
}
else
{
// form submitted

// where is the socket server?
$host="192.168.1.99";
$port = 1234;

// open a client connection
$fp = fsockopen ($host, $port, $errno, $errstr);

if (!$fp)
{
$result = "Error: could not open socket connection";
}
else
{
// get the welcome message
fgets ($fp, 1024);
// write the user string to the socket
```

```
fputs ($fp, $message);
// get the result
$result .= fgets ($fp, 1024);
// close the connection
fputs ($fp, "END");
fclose ($fp);

// trim the result and remove the starting ?
$result = trim($result);
$result = substr($result, 2);

// now print it to the browser
}
?>
Server said: <b><? echo $result; ?></b>
<?
}
?>

</body>
</html>
```

Different Strokes

If you'd prefer to, there's also an alternative, somewhat longer approach to constructing a client. Most of the time, you won't need to use this − fsockopen() is more than sufficient for most requirements − but it's included here for reference purposes. Take a look at this next script, which replicates the functionality of the previous example:

```
<html>
<head>
</head>

<body>

<?
// form not yet submitted
if (!$submit)
{
?>
<form action="<? echo $PHP_SELF; ?>" method="post">
Enter some text:<br>
<input type="Text" name="message" size="15"><input
type="submit"
name="submit" value="Send">
</form>
<?
}
else
{
// form submitted

// where is the socket server?
$host="192.168.1.99";
$port = 1234;

// create socket
$socket = socket_create(AF_INET, SOCK_STREAM, 0) or die("Could
not create
socket\n");

// connect to server
$result = socket_connect($socket, $host, $port) or die("Could
not connect
to server\n");

socket_read ($socket, 1024) or die("Could not read server
response\n");
```

```
// send string to server
socket_write($socket, $message, strlen($message)) or
die("Could not send
data to server\n");

// get server response
$result = socket_read ($socket, 1024) or die("Could not read
server
response\n");

// end session
socket_write($socket, "END", 3) or die("Could not end
session\n");

// close socket
socket_close($socket);

// clean up result
$result = trim($result);
$result = substr($result, 0, strlen($result)-1);

// print result to browser
?>
Server said: <b><? echo $result; ?></b>
<?
}
?>

</body>
</html>
```

In this case, the socket_connect() function is used to open a connection to the server, with the familiar socket_read() and socket_write() functions used to receive and transmit data over the socket connection. Once the result string has been obtained from the server, the socket connection is closed with socket_close() and the output is printed to the browser.

Again, this is an alternative implementation − it's unlikely that you'll find much use for it, as the fsockopen() function provides a much simpler (and shorter) way to accomplish the same thing.

POP Goes The Weasel

Let's try something a little more advanced. How about a TCP client that connects to a POP3 server to retrieve the total number of messages for a user's mailbox?

In order to build such a client, I need to first understand the sequence of commands passed to a POP3 server in order to obtain the message total, and then replicate this sequence in my PHP−based client. The best way to do this is by using a regular telnet client to interact with the server and understand the command sequence − so here goes:

```
$ telnet mail.host 110
Trying 192.168.0.1...
Connected to 192.168.0.1.
Escape character is '^]'.
+OK POP3 mail.host v5.5 server ready
USER john
+OK User name accepted, password please
PASS doe
+OK Mailbox open, 72 messages
STAT
+OK 72 24595628
QUIT
+OK Sayonara
Connection closed by foreign host.
```

As you can see from the sample session above, the second element of the string returned by a STAT command

```
STAT
+OK 72 24595628
```

holds the total number of messages (72, in this case). All that's needed, therefore, is a script that connects to the POP3 server (usually available on port 110), sends the sequence of commands above, retrieves the output of the STAT command, and extracts the message total from it.

Here's the script to accomplish this:

```
<?
// mail server settings
$host="192.168.0.99";
$port = 110;
$user = "john";
$pass = "doe";
```

```
// open a client connection
$fp = fsockopen ($host, $port, $errno, $errstr);

// if a handle is not returned
if (!$fp)
{
die("Error: could not open socket connection\n");
}
else
{
// get the welcome message
$welcome = fgets ($fp, 150);

// check for success code
if (substr($welcome, 0, 3) == "+OK")
{
// send username and read response
fputs ($fp, "USER $user\n");
fgets($fp, 50);

// send password and read response
fputs ($fp, "PASS $pass\n");
$ack = fgets($fp, 50);

// check for success code
if (substr($ack, 0, 3) == "+OK")
{
// send status request and read response
fputs ($fp, "STAT\n");
$status = fgets($fp, 50);
if (substr($status, 0, 3) == "+OK")
{
// shut down connection
fputs ($fp, "QUIT\n");
fclose ($fp);
}
// error getting status
else
{
die ("Server said: $status");
}
}
// auth failure
else
{
die ("Server said: $ack");
}
}
// bad welcome message
```

```
else
{
die ("Bad connection string\n");
}

// get status string
// split by spaces
$arr = explode(" ", $status);

// the second element contains the total number of messages
echo $arr[1] . " messages in mailbox";
}
?>
```

And here's the output:

```
$ /usr/local/bin/php -q popclient.php
72 messages in mailbox
```

How does this work? Very simple.

First, a connection is opened to the POP3 server using the fsockopen() function discussed previously; the arguments to this function (host, port et al) are obtained through PHP variables which are set at the top of the script.

```
<?
// open a client connection
$fp = fsockopen ($host, $port, $errno, $errstr);
?>
```

Once a socket connection has been established, the fgets() and fputs() functions are used to send POP3 commands to the server via this socket, and read the resulting output into PHP variables.

```
<?
// send username and read response
fputs ($fp, "USER $user\n");
fgets($fp, 50);

// send password and read response
fputs ($fp, "PASS $pass\n");
$ack = fgets($fp, 50);
?>
```

A POP3 server typically prefixes the result of every successful command with the string "+OK". This knowledge allows for simple error−checking within the script − note how the output of fgets() is checked at every stage, with subsequent commands executed only if the previous one was successful.

```
<?
// send status request and read response
fputs ($fp, "STAT\n");
$status = fgets($fp, 50);
if (substr($status, 0, 3) == "+OK")
{
// shut down connection
fputs ($fp, "QUIT\n");
fclose ($fp);
}
// error getting status
else
{
die ("Server said: $status");
}
?>
```

Once the output of the STAT command has been received, the socket is closed, and the result string is split into its constituent parts with PHP's very cool explode() function. The message total is then extracted and printed.

```
<?
// get status string
// split by spaces
$arr = explode(" ", $status);

// the second element contains the total number of messages
echo $arr[1] . " messages in mailbox";
?>
```

Simple, huh?

Access Denied

Here's another example, this one setting up an authentication server that accepts a username and password and verifies them against the standard Unix /etc/passwd file. Take a look:

```
<?

// don't timeout!
set_time_limit(0);

// set some variables
$host = "192.168.1.99";
$port = 1234;

// create socket
$socket = socket_create(AF_INET, SOCK_STREAM, 0) or die("Could
not create
socket\n");

// bind socket to port
$result = socket_bind($socket, $host, $port) or die("Could not
bind to
socket\n");

// start listening for connections
$result = socket_listen($socket, 3) or die("Could not set up
socket
listener\n");

// accept incoming connections
// spawn another socket to handle communication
$spawn = socket_accept($socket) or die("Could not accept
incoming
connection\n");

// read client input
$input = socket_read($spawn, 1024) or die("Could not read
input\n");

// clean up input string
$input = trim($input);

// split input into components and authenticate
$arr = explode(":", $input);
$result = authenticate(trim($arr[0]), trim($arr[1]));

socket_write($spawn, $result, strlen ($result)) or die("Could
```

```
not write
output\n");

// close sockets
socket_close($spawn);
socket_close($socket);

// authenticate username/password against /etc/passwd
// returns: -1 if user does not exist
// 0 if user exists but password is incorrect
// 1 if username and password are correct
function authenticate($user, $pass)
{
$result = -1;
// make sure that the script has permission to read this file!
$data = file("/etc/passwd");

// iterate through file
foreach ($data as $line)
{
$arr = explode(":", $line);
// if username matches
// test password
if ($arr[0] == $user)
{
// get salt and crypt()
$salt = substr($arr[1], 0, 2);
if ($arr[1] == crypt($pass, $salt))
{
$result = 1;
break;
}
else
{
$result = 0;
break;
}
}

}
// return value
return $result;
}

?>
```

Most of this should now be familiar to you, so I'm not going to get into the details of the socket connection itself; I will, however, briefly explain how the authentication is carried out.

In this case, the client is expected to provide a username and (cleartext) password in the format "username:password" to the server over the socket connection. The server then reads the system's password file (usually /etc/passwd or /etc/shadow), looks for a line beginning with the specified username, and extracts the first two letters of the corresponding encrypted password string. These two characters serve as the "salt" for the encryption process.

Next, the cleartext password is encrypted with PHP's crypt() function and the extracted "salt", with the result checked against the encrypted value in the password file. If the two match, it implies that the supplied password was correct; if they don't, it implies that the password was wrong. Either way, the result of this authentication procedure is then returned to the client over the socket connection.

Here's the output of a session with this server:

```
$ telnet 192.168.1.99 1234
Trying 192.168.1.99...
Connected to 192.168.1.99.
Escape character is '^]'.
john:doe
1Connection closed by foreign host

$ telnet 192.168.1.99 1234
Trying 192.168.1.99...
Connected to 192.168.1.99.
Escape character is '^]'.
nosuchuser:hahaha
-1Connection closed by foreign host
```

Game Over

And that's about all I have time for. In this article, you learned a little bit about PHP's socket programming functions, using them to create and manipulate socket connections in a networked environment. In addition to some simple examples, you also learned how to apply PHP's socket API to real−life situations with practical examples, including a POP3 client and an authentication server.

The latest version of PHP, PHP 4.1.0, comes with an updated socket API, one which offers developers greater control over socket creation and closely mimics the socket API used by programming languages like C and C++. As of this writing, many of these new functions have not yet been documented in the PHP manual; however, if you're familiar with socket programming in C, you should have no trouble adapting your code to use these new functions.

You can read up on PHP's socket functions at http://www.php.net/manual/en/ref.sockets.php

If you'd like to learn more about sockets and socket programming in general, here's a list of sites you might want to consider visiting:

http://www.ecst.csuchico.edu/~beej/guide/net

http://www.lcg.org/sock−faq

http://www.onlamp.com/pub/a/php/2001/03/29/socket_intro.html

http://phpbuilder.net/columns/armel20010427.php3?print_mode=1

http://w3.softlookup.com/tcp_ip

I hope you found this article interesting, and that it helped open your eyes to one of PHP's lesser−known capabilities. Let me know if you liked it...and, until next time, stay healthy!

Note: All examples in this article have been tested on Linux/i386 with Apache 1.3.12 and PHP 4.1.0. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!