CHAPTER 8

Daemons

In reading this chapter, you might get the idea that I'm telling you to use PHP to build large-scale, daemon-based applications. While I definitely am of the opinion that PHP can be used for this purpose, the language was not designed for it. For me, the question of whether PHP can be used as a daemon comes out of the "use the best tool for the job" discussion that often happens online. Usually, this topic comes up as an argument after one person bested another in some kind of language shootout. The looser of the argument (regardless of the actual merits of his or her chosen language) half-concedes by saying, "Well, you just use the best tool for the job." Meaning, "You got me, but I won't be made to look the fool." Or, "It's not worth arguing with this idiot." Chances are it's the latter.

The problem is that there are lots of jobs that need to be done that are outside the specialty of a given language. Following the "best tool for the job" mentality, you might have part of your infrastructure running PHP, part of it running Ruby on Rails, part of it running .NET, and part of it running Java. I've got nothing against any of those languages, but absolutely *none* of them do *everything* well.

Java is a good example. I personally like programming in Java. However, to have a Web site running in Java, you need to have layer upon layer upon layer of application server upon application server upon application server to make it run. Will it run? Yes. Will it run fast? Yes. Are Web pages what Java was designed to do? With due respect to JavaServer Pages (JSP) developers, no.

Same thing with Ruby. Ruby is really good if you can afford to have very little flexibility. The benefit of Ruby on Rails comes at the price of inflexibility. If you

need specific control over aspects of your application, Ruby is probably not the best choice for you.

Where am I going with this discussion? The question of using "the best tool for the job." A heterogeneous environment is not a good one to build for or to manage. Getting one of those to work should be the goal of academics. Real life, with all its warts, does much better with a fuller understanding of a smaller subset of features. This is because as soon as you have more than one type of architecture in your organization, you need to have more than one skill set. Having more skill sets means it's hard to find experts. And fewer experts means more time spent on support calls with other people whose knowledge base is also too wide.

What's the solution? I would contend that there is not one. Just as there is no vehicle that serves all needs, there is no programming language that serves all needs. In addition, the people you have around are important factors. If you have developers who are good, but not great, at PHP, Ruby, Python, Java, and .NET (who also need to know Flash), where will you have the skill set internally to handle a problem for which you can't find an answer on Google? It's like buying a Peugeot in Wyoming. I believe that if you have several languages and infrastructures that you support, your developers are going to have more problems and be less creative.

Creativity for the sake of creativity is not good. However, creativity enables innovative solutions to difficult problems. Far too many people pick up a language and work with it, thinking that they are experts, because someone else said the language was cool. However, the more languages you know, the less of an expert you will be in each.

This takes us back to the earlier statement about using "the best tool for the job." I would contend that this philosophy is wrong, or at least, problematic. I'd go the route of saying "use the best tool for the organization" instead. That organization might be your place of business, a nonprofit you work for, the Web site of a friend who's starting a new business, your church, anything. How do you decide which tool you are going to use for a specific problem?

What you base the decision on is not set in stone. In fact, it's unlikely that your organization made an intentional decision beyond, "Hey, we're building a Web site;

we should use PHP, right?" If your organization has a significant Web presence, then that is probably the right call. PHP does Web good. But what if you need something else? Say, for example, you need a simple message queue. You have several options. One is to obtain some kind of third-party messaging software. There are plenty of options available for you, and they probably are going to be better than what you would build.

But there is often a problem with using off-the-shelf software, both proprietary and open source. Half the time, it's complicated enough that once you install it, you need even more experts to manage it. So, say you have simple needs. You need a message queue, but your organization does not have the skill to support another language.

That is where building a PHP daemon comes in.

Starting Out

If we're going to be building a daemon, one of the primary goals is to build something that can handle multiple tasks at one time. To do that, we need to be able to handle multiple connections at once. Building an application that can handle multiple connections at once is actually very simple. So simple, in fact, that we've already done it, back in the chapter on networking and sockets. However, we have two problems, one old and one new. Both deal with how to use the resources on the system in the best way possible.

The old problem is the question of how to efficiently use CPU time when we may have significant parts of our work that require other services and, thus, have wait times. These can be tasks such as database calls, network calls, or even file system access. Even though disks are relatively fast these days, I/O contention does occur. This isn't just a problem that happens for larger organizations with high performance environments; smaller shops that have inefficient data structures on disk or just high-performance data requirements also face this issue. So, how do we do other things while waiting for slower things to do their thing?

The newer problem is probably one with which you're already familiar. With processors opting now for multi-core, multi-CPU, hyper-threaded architectures, a big issue to solve is that of using all of those CPUs and cores. PHP has a simple

architecture that is great for HTTP. But it doesn't work so well once you start building a daemon-ized architecture. PHP has no internal mechanism for doing things asynchronously. Why should it? In PHP's native arena, Apache handles that work. And even the default **prefork** Multi-Processing Module (MPM) in Apache doesn't really do a lot of asynchronous processing because each process is separate from the others and handles only one request at a time.

So the first problem we need to solve is how to handle doing more than one thing at a time. There are two ways to do this, and the method you choose will depend on what type of system you're working on. If you are building a daemon, handling many incoming connections, then using a forking, or rather, a pre-forking, method is likely the best option.

In our prior examples using non-blocking I/O, we had to have a relatively complex piece of logic to handle data coming in from multiple sources. When you use pre-forking, this is not required, depending on how you do it. That's because forking actually lets the operating system manage the connections instead of you doing it yourself.

When a process is forked, the system copies the allocated resources into a new segment of memory, and both processes are resumed after the **fork()** call. The only difference between the two is the return value of the call. The parent process receives the process ID of the child, and the child receives a process ID of **0** (zero). If an error occurs, **-1** is returned.

Because the process being copied is mostly identical, some shared resources remain somewhat shared. An example of this is a socket that was opened prior to the forking. If a socket was created before the forking, you likely do not need to worry about non-blocking I/O. That is because existing connections are duplicated during the forking operation. When a new socket is accepted, the socket is unique to the individual forked process, even though the server socket is shared. Each forked process represents a resource that is available to do a specific unit of work. Because blocking operations on one work unit will not affect the operations on another work unit we can build our functionality without having to come up with an architecture that is tolerant of wait times on disk and network I/O.

To handle multiple work units, any solution we implement will likely need to work similarly to the pre-forking method. But there is, of course, the problem of how to do that on Windows. Windows does not support forking. To get around this limitation, we would need to create a daemon to act as a broker between the incoming requests and the worker processes that will ultimately handle the work. That type of broker would require very few CPU resources and would need to run as a single process/thread. Clearly, PHP doesn't do this, but the broker functionality is relatively simple, and as such it wouldn't matter that it uses only a single core. The operating system would take care of allocating the other cores automatically because they would be distinct processes from each other.

The second problem is going to be how to share data. But before examining that challenge, let me state something that you should keep in the back of your mind. Try to *not* share data, especially writeable data. That is a tall order, indeed. However, sharing data, particularly sharing data that needs to be written to, is one of the Achilles' heels of modern programming. It is far too easy to read data that is in the process of being overwritten. That is not to say that you shouldn't share data, but only that you should proactively minimize the use of shared resources. Consider defining as many resources as you can as read-only.

Also, try to write your data historically rather than overwriting values. When you overwrite data, you generally need to lock the resource. Fine-grained locks are not desirable. Locks are often necessary, but try to minimize them.

Another thing to consider is loading as many of your resources up front as possible. Consistency can be a problem in a shared environment. By front-loading as much data as you can, you reduce the likelihood of a collision between current and out-of-date data in the same data set. Inconsistency is probably a bigger threat than wasting some compute time processing old data.

As a language, PHP is very good at serialization, while other languages tend to restrict you more. The purpose of this restriction in other languages is mostly for security. Serializing data exposes that data to the outside world. It is quite possible that the data in question may be available in a context in which there are no security considerations to be had. Myself, I am somewhat unconvinced that this restriction truly buys a solid layer of security — the reason being that if your system is compromised to the point where an attacker can read serialized data,

your problems are probably bigger than an attacker being able to read serialized data.

Using data serialization, you can easily pass virtually any data between PHP processes. One exception to this rule would be anything that has a resource variable in it. So things such as file handles or database connections cannot be passed. But passing regular serialized data is relatively easy and can be done by using a very simple protocol. Basically, you send four bytes (or eight bytes) that state the length of the serialized object and then send the raw object. Sometimes, the binary data in a serialized object can be problematic for some mechanisms, but as long as you use **fread()**, **fwrite()**, and so on in combination with sending the serialized data length, you should be fine. This point is important because some functions may not like reading the binary content because serialization operations can generate null characters which could be interpreted as the end of the string.

One of the benefits of serialization is that it lets you pass data within the context of your entire application. Say you need to pass information about a user. Yes, you could use Soap and have all the mappings done between the Soap call and native code, but passing serialized data is just nice, efficient, and native. Modern applications seem to be sold on how many layers are required to make it run. There's something about "here's your frigging data" that's kind of refreshing as many of the problems I've seen are due to overly complex application layers.

These are all points that you will have to take into consideration if you are to build your own PHP daemon. With PHP 5.3's new garbage collection mechanism, the walls preventing long-running PHP daemons are slowly being torn down. The last major wall is that of architecting your application to make full use of your CPU resources — in other words, how to do more than one thing at a time.

Our Example

For simplicity's sake, I am going to focus on building a pre-forking daemon, similar in approach to the Apache **prefork** MPM. It is rather unlikely that PHP developers deploying to Windows are going to run PHP as a daemon

any time soon. Linux developers are more likely to do that, even though they are only slightly less unlikely to do it. So, with apologies to all the PHP developers on Windows (and there are a bunch), this example will focus on Linux. The same thing can be achieved on Windows using a proxy architecture similar to FastCGI.

Our example will be a simple spider. It will run in the background while PHP runs on the Web server. When someone makes a request stating the desire to spider a Web site, the PHP on the Web server will create an object representing that request and pass it off to the daemon, which will then handle the spidering of the Web site.

The first class definition is for the **Url** class, which will be used by both the front end and the back end to initiate the spider request. The front end uses it to pass the basic information about the URL to the back end and then disconnect. The back end uses it to spawn the request for each individual URL it finds on the site.

Figure 8.1 shows the code to define the **Url** class.

```
class Url
{
      private $_url;
      private $_recursive;
      private $_disconnect;

      public function __construct(
            $url,
            $recursive        = false,
            $disconnect       = true
      )
      {
            $this->_url             = $url;
            $this->_recursive       = $recursive;
            $this->_disconnect      = $disconnect;
      }

      public function getUrl()
      {
            return $this->_url;
      }
```

```
     public function isRecursive()
      {
            return $this->_recursive;
      }

      public function willDisconnect()
      {
            return $this->_disconnect;
      }
}
```

Figure 8.1: URL container class

For handling the responses, we have a simple class, **UrlResponse** (Figure 8.2), that will be used to pass data from a worker back to the dispatcher process.

```
class UrlResponse
{

      private $_body;
      private $_links = array();

      public function __construct($body,
                                  array $links = array())
      {
            $this->_body = $body;
            $this->_links = $links;
      }

      public function getBody()
      {
            return $this->_body;
      }

      public function getLinks()
      {
            return $this->_links;
      }
}
```

Figure 8.2: Data class for holding the response from a URL

Our front end will consists of a single form (Figure 8.3). I use **Zend_Form** because the actual form HTML is kind of irrelevant, and the code used here should be easy to conceptualize even if you are not familiar with Zend Framework.

```
class UrlForm extends Zend_Form
{

      public function init()
      {
            $this->setMethod('POST');
            $this->addElement(
                  'text',
                  'url',
                  array('label' => 'URL')
            );
            $this->addElement(
                  'checkbox',
                  'recursive',
                  array(
                        'label'=> 'Do recursive spider'
                  )
            );
            $this->addElement(
                  'submit'
            );
      }
}
```

Figure 8.3: A Zend_Form based class for submitting the URL

When it is submitted, the PHP script on the server side will serialize the URL object and submit it to the back-end daemon for processing (Figure 8.4).

```
$form = new UrlForm();
$form->setView(new Zend_View());

if ($_SERVER['REQUEST_METHOD'] === 'POST'
      && $form->isValid($_POST)) {
      $url = new Url(
            $form->getValue('url'),
            $form->getValue('recursive'),
            true
      );
```

```
      $serverSock = fsockopen('tcp://dev', 10000);
      $data = serialize($url);
      $data = pack('N', strlen($data)) . $data;
      fwrite($serverSock, $data);
      fflush($serverSock);
      echo "URL submitted";
} else {
      echo $form;
}
```

Figure 8.4: Code to submit the URL to the daemon socket

We take the data from the form submission and create a new **Url** object. From there, we serialize it and then build the data stream by prepending the size as a 32-bit unsigned long. You will notice that we build the entire string instead of sending it in pieces. That is because we don't want to give the network the opportunity to flush bits of data. The purpose of this example is to write a decently scalable networking application without the connection management of non-blocking I/O. For this goal, we need to be able to read the full string from the stream. Making two write calls can cause separate packets to be sent. In the Web–PHP portion of the script, this didn't seem to be much of an issue, but on the command-line interface (CLI) it is.

Now for the fun stuff. The daemon will be a class called **Daemon** (no points for creativity). It will contain an interface to the **Zend_Search_Lucene** search engine in Zend Framework. I used this solution for the simple reason that it was already written. Figure 8.5 shows the starting definition for the **Daemon** class.

```
class Daemon {

      private $_sSock;
      private $_workerCount = 0;

      /**
       *
       * @var Zend_Search_Lucene_Interface
       */

      private $_index;

}
```

Figure 8.5: Base Daemon class

In here, we have a property that holds the server socket resource, a property that holds the count for the number of forked processes we'll start up, and then the object for the index. When a connection is made to the daemon from the Web server, the pre-forked process that answers the request manages the spidering of the Web site, farming out the individual HTTP requests to other pre-forked processes while taking the results and storing them in the Lucene index. The "answering" process loads the first page, retrieving all the URLs on the page that match the current docroot and passing them off to individual workers. That is defined in a function called **execute()** (Figure 8.6).

```
public function execute($host, $port, $workerCount)
{

  $this->_index = Zend_Search_Lucene::create(
        '/tmp/index'
  );
  $this->_workerCount = $workerCount;
  $this->_sSock = socket_create(
        AF_INET,
        SOCK_STREAM,
        SOL_TCP
  );
  socket_set_option(
        $this->_sSock,
        SOL_SOCKET,
        SO_REUSEADDR,
        1
  );

  if (!socket_bind($this->_sSock, $host, $port)) {
       throw new Exception("Unable to bind socket");
  }

  if (!socket_listen($this->_sSock)) {
      throw new Exception("Unable to listen on socket");
  }

  for ($c = 0; $c < $workerCount; $c++) {
        if (($pid = pcntl_fork()) === 0) {
              $this->_execute();
        } else {
              echo "Child {$pid} started...\n";
        }
  }

  socket_close($this->_sSock);
  pcntl_wait($status);
}
```

Figure 8.6: Code to execute the daemon

This method primarily does two things: it creates the server socket, and it pre-forks the correct number of processes.

Creating the socket is similar to what we saw in the networking and sockets chapter, but we have an additional function call here to **socket_set_option()**, setting the socket option **SO_REUSEADDR** to **1**. The purpose of this step is so that if the daemon goes down without properly closing the server socket, we can immediately bring it back up. When a daemon goes down and the socket is still open, the socket will be in a **TIME_WAIT** state if there are clients attached to it. What this means is that the socket has closed, and the client side connection has closed, but the socket is waiting for a timeout to occur. This is normal TCP behavior. However, by default, Linux will not let you bind to a socket that's listening on the same port as a socket in the **TIME_WAIT** state until the socket times out and is removed. By setting **SO_REUSEADDR**, we can re-bind to that socket while it is still in a **TIME_WAIT** state. If it is in the **LISTEN** or **ESTABLISH** state, it will not bind. Only if it is in **TIME_WAIT** or if the socket has timed out can we re-bind.

The second thing the **execute()** method does is perhaps the most important part of our discussion. It occurs in the **for()** loop. Here, we will create all the individual pre-forked processes. When we call the **pcntl_fork()** process, Linux takes the memory allocated to the application and copies it to a new memory space. This includes objects, classes, sockets, and so on. The return value of **pcntl_fork()** will differ depending on whether you are the child process or the parent process. If you are the child, it will return zero. If you are the parent, it will return the parent of the child.

After that, we close the socket for the parent process and sit on **pcntl_wait()**. The reason we close the socket is because we don't want this process to answer any connections, and we want to keep it in the foreground so we can easily kill the process for testing reasons. The **pcntl_wait()** function call will wait until one of the children receives some kind of signal, such as an interrupt or a kill. We could actually use this mechanism to manage the children, however in this example we're just using it to keep the parent process from dying. With this code, killing just one child process will cause the parent process to exit because **pcntl_wait()** is called only once. One of the things we could do is have it sit on **pcntl_wait()** and if a child goes down have it spawn another one. But I didn't want to do a whole bunch of process management stuff because that can get to be a little confusing, particularly in print. Calling **pcntl_wait()** just once lets the parent process hang around so we

can simply hit **Ctrl-C** and have the parent, plus all the children, terminate in one action.

Once a process has been forked, and it is a child, we need to be able to manage the incoming connections. The **_execute()** method call handles that work (Figure 8.7).

```
private function _execute()
{
      while (true) {
            $sock = socket_accept($this->_sSock);

            $obj = $this->_readData($sock);
            if (!$obj instanceof Url) {
                  socket_close($sock);
                  continue;
            }
            echo 'PID '
                   . posix_getpid()
                   . ' received URL '
                   . $obj->getUrl()
                   . "\n";

            try {
                  $url = Zend_Uri_Http::fromString(
                        $obj->getUrl()
                  );
                  if ($obj->willDisconnect()) {
                        socket_close($sock);
                  }
                  $reqData = @file_get_contents($url);
                  $this->_processPage(
                        $obj,
                        $url,
                        $reqData,
                        $sock
                  );

            } catch (Exception $e) {
                  echo $e->getMessage() . "\n";
            }
            if ($sock && is_resource($sock)) {
                  socket_close($sock);
            }
      }

}
```

Figure 8.7: Code to handle child process requests

As you can see, we put ourselves into an infinite loop. We then sit on a **socket_accept()** call on the socket that we created earlier. This is the primary method of handling multiple connections. Because we copied the memory from the parent process after creating the socket, the kernel will handle dispatching the request to each individual child process as a new connection on the server socket comes in. From there, we call the **_readData()** method, which reads the initial data from the socket. We'll see that step in a bit.

Because the request must be made with a serialized version of **Url**, if the data is not an instance of **Url** then we simply close the connection. After printing some informational data (which technically should go to a log file), we parse the URL to make sure it is valid. Then, we check to see whether the client cares about the response, by calling **willDisconnect()**. An example of this would be the front-end Web server. The Web server does not want to hang around while a site is spidered. And so, the daemon immediately disconnects once it has read and parsed the request. On the other hand, the dispatch process that manages the spider operation will want to stay around so it can retrieve the body and the list of links from the worker process.

Having done all that, we then call **file_get_contents()** on the URL and pass all that data on to the **_processPage()** method. However, before we look at the **_processPage()** method, let's take a quick look at **_readData()** (Figure 8.8).

```
private function _readData($sock)
{
      $data = socket_read($sock, 4);
      if (!$data) {
            return false;
      }

      $data = unpack('N', $data);
      $len = array_shift($data);
      $data = socket_read($sock, $len);
      $obj = @unserialize($data);

      return $obj;
}
```

Figure 8.8: Reading data from the server socket

This is really a very simple method. We know that the first four bytes are a 32-bit long unsigned integer. So we read four bytes and unpack that data. We then know exactly how much data we need to read from the socket when we call **socket_read()**. Because we're using blocking I/O and letting the kernel handle the asynchronous operations, the code is much simpler than it would be had we used non-blocking I/O. After we read the data, we unserialize it. If the data we read is malformed, **unserialize()** will throw an error and return **false**. Good or bad, we return the value.

Pretty easy. Now that we have that operation out of the way, let's look at the **_processPage()** call (Figure 8.9).

```
private function _processPage(
     Url $obj,
     Zend_Uri_Http $url,
     $reqData,
     $sock
)
{
     $links = $this->_getLinks(
           $reqData,
           $url->getHost(),
           $url->getPath()
     );
     $farmedLinks = array();
     foreach ($links as $link) {
           $farmedLinks[$link] = false;
     }
     // Don't re-spider the original URL
     $farmedLinks[$obj->getUrl()] = true;
     if (!$obj->willDisconnect()) {
           $response = new UrlResponse(
                 $reqData,
                 $links
           );
           $data = serialize($response);
           $data = pack('N', strlen($data)).$data;
           socket_write($sock, $data);
           socket_close($sock);
     }
     if ($obj->isRecursive()) {
           $this->_farmLinks($farmedLinks);
           $this->_index->commit();
```

```
            echo "Data received.  Optimizing index...\n";
            $this->_index->optimize();

            echo "Optimizing Complete.\n";
        }
}
```

Figure 8.9: Indexing a page and gathering links

The first thing **_processPage()** does is call the **_getLinks()** method, which uses XML Path Language (XPath) to query the HTML page. We'll look at that step shortly. From there, **_processPage()** captures all the links and places them in an associative array so that they can be returned as part of the **UrlResponse** object. This is only done if the client intends to disconnect. The front-end request will disconnect. But if for some reason it doesn't want to, this mechanism could be used to report back the links that were found on the page. However, it is most likely to be used by the dispatch process so it can process the links and hand them off to other worker processes.

If the request is a recursive request, meaning a full spider of the URL, the method passes off the links to a method called **_farmLinks()**, which is where the actual dispatching occurs.

The **_getLinks()** method (Figure 8.10) is a relatively long method, but it's primary purpose is to get all the **href** attributes and return fully formatted URLs. That's where most of the length comes in.

```
private function _getLinks($data, $host, $path)
{
   $currentDir = dirname($path);

   $doc = new DOMDocument();
   @$doc->loadHTML($data);
   $links = array();
   $xPath = new DOMXPath($doc);
   $nodeList = $xPath->query('//a[@href]');
   foreach ($nodeList as $node) {
     $link = $node->getAttribute('href');
```

```
    if (!$link) $link = $path;
    if (stripos($link, 'http://') === 0) {
         if (stripos($link,'http://'.$host.'/')!== 0) {
              continue;
         }
    } else if (stripos($link, 'https://') === 0) {
         if (stripos($link,'https://'.$host.'/')!== 0){
              continue;
         }
    }
    if (($ancPos = strpos($link, '#')) !== false) {
         $link = substr($link, 0, $ancPos);
    }

    if (strlen($link) >= 2
         && $link[0] == '.'
         && $link[1] == '/') {

         $link = substr($link, 2);
    }
    if (!$link) {
         continue;
    } else if ($link[0] === '/') {
         $link = 'http://'
                    . $host
                    . $link;
    } else if (stripos($link, 'http://') !== 0
                && stripos($link, 'https://') !== 0) {
         $link = 'http://'
                    . $host
                    . $currentDir
                    . '/'
                    . $link;
    }

    if (array_search($link, $links) === false) {
         try {
              $l = Zend_Uri_Http::fromString($link);
              $links[] = $link;
         } catch (Exception $e) {}
    }
  }

  return $links;
}
```

Figure 8.10: Retrieving all the links from a page

We make a good effort to format the data as best we can. Next, we check to see whether we already have that link found. Then, we do a final check using **Zend_Uri_Http** to make sure that the format is correct. If we were not able to generate a properly formatted URL, we simply ignore it via the empty **catch** statement.

The **_farmLinks()** method (Figure 8.11) takes the links that were supplied and sends them to the worker connections to do the actual downloading and parsing of content for the remote page.

```
private function _farmLinks($farmedLinks)
{

  $connCount = (int)($this->_workerCount / 3);

  $addr = $port = null;
  socket_getsockname($this->_sSock, $addr, $port);

  $req = array();
  $write = $except = array();
  while (($link = array_search(false, $farmedLinks))
               !== false
               || count($req) > 0) {

        if ($link !== false) {
              $farmedLinks[$link] = true;
              $sock = $this->_sendPacket(
                   'localhost',
                   $port,
                   $link
              );
              $req[$link] = $sock;
        }
        $timeout = 0;
        if (count($req) >= $connCount
             || array_search(false, $farmedLinks)
                === false) {
              $timeout = null;
        }
        $conn = $req;
        socket_select($conn, $write, $except, $timeout);
        foreach ($conn as $c) {

          $obj = $this->_readData($c);
```

```
          $connKey = array_search($c, $req);
           socket_close($c);
           unset($req[$connKey]);
           $farmedLinks[$connKey] = true;
           if ($obj instanceof UrlResponse) {

             $doc =
              Zend_Search_Lucene_Document_Html::loadHTML(
                 $obj->getBody()
                 );
                 $doc->addField(
           Zend_Search_Lucene_Field::keyword(
                  'url',
                  $connKey
             )
                 );
                 $this->_index->addDocument($doc);

                 foreach ($obj->getLinks() as $link) {
                       if (!isset($farmedLinks[$link])){
                             $farmedLinks[$link]=false;
                             unset($req[$link]);
                       }
                 }
           }
         }
   }
}
```

Figure 8.11: Sending the requests to parse the links to child workers

The first thing the **_farmLinks()** method does is determine how many workers it is going to use. In this case, we choose a third of them, although as long as you choose a fraction that is at least a few less than the number of connections you should be fine. The call to **socket_getsockname()** is there only so we can get the actual port on which the server is listening.

After that, it's time to start iterating over the results. We do this by trying to find the first instance in **$farmedLinks** whose value is false. Earlier, we iterated over all the links, changing the values (which were the URLs) to be the keys and setting the **$link** value to false for each one. False values would mean that the URL had not been spidered yet, whereas true values would mean that it had. So we retrieve the next URL or, if there are no more links to be found, check to see whether we have

some connections open. If either of these conditions is true, we iterate over the array, sending data to the workers.

If **$link** is not false, it means that we found a link that has not been spidered yet. In that case, we initiate a connection by calling **_sendPacket()**. This quick and easy helper function is shown in Figure 8.12.

```
private function _sendPacket($host, $port, $link)
{
   $sock = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
   socket_connect($sock, $host, $port);
   $url = new Url($link, false, false);
   $data = serialize($url);
   $data = pack('N', strlen($data)) . $data;
   socket_write($sock, $data);
   return $sock;
}
```

Figure 8.12: Code to send the response back

The function's sole purpose is to take the URL and place it into a **Url** object, connecting to a worker process and sending the serialized **Url** object. After sending the object, **_sendPacket()** takes the socket that was created and returns it so that the **_farmLinks()** method can manage it.

Once we have that socket back, we place it into the **$rec** array so we can manage it later on. The next thing we need to do is determine what our read timeout is. One thing we do not want to do is poll TCP connections. That would mean we would have to do a read on each socket, let it time out, and go on to the next one — a very inefficient process. Instead, we use the **socket_select()** function, just as we did with non-blocking I/O, and have it select on the sockets that are in the **$rec** array.

If data has been received on any of the sockets, we can call the **_readData()** method on the socket that was returned as part of **$conn**. This is a blocking call, but because the socket was returned from the select operation, we know that there will be data on it, so no read timeout is needed. If a data stream were longer than the TCP packet size, a delay could occur. On a remote machine on an Ethernet

network, that size, called the Maximum Transmission Unit (MTU), is 1,500 bytes. However, we are connecting over the local interface, so in our case any lag would be negligible. In addition, the MTU for the local interface is usually larger than an Ethernet one. Mine is 16,436 bytes. If we were doing this over a larger network, non-blocking I/O might be more pertinent, but here it is not. Another reason for opting to connect over the local interface is because I wanted a simpler example than what we would have had to do with non-blocking I/O over multiple connections on a pre-forked server. It's moderately easy to do, but also very easy to get lost if it is a new concept. So I sacrificed a small increase in efficiency for clarity of code.

Once we receive and process the data, we close the socket, unset the variable in **$req**, and set the value to true in **$farmedLinks** so we don't download the data again. If we get a response back of **UrlResponse**, that means that the request was successful and we can now place the URL into our Lucene index. We do that by creating a document object of a type that was specifically created for HTML and adding an additional field so we can get the URL for search results.

That encompasses most of our functionality. The last thing we need to do is create the **Daemon** object and execute it. Figure 8.13 shows how simple this is to do.

```
require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance()
                        ->setFallbackAutoloader(true);

$d = new Daemon();
$d->execute('0.0.0.0', 10000, 10);
```

Figure 8.13: Kicking off the daemon

Because we are using several Zend Framework components, we make sure to load the autoloader and then execute the **Daemon** object, telling it to listen on all interfaces, on port 10000, with 10 workers.

Does it work? Figure 8.14 shows the output generated when we start the daemon.

```
[apache@localhost Daemon]#
  /usr/local/zend/bin/php daemon.php
Child 3000 started...
Child 3001 started...
Child 3002 started...
Child 3003 started...
Child 3004 started...
Child 3005 started...
Child 3006 started...
Child 3007 started...
Child 3008 started...
Child 3009 started...
```

Figure 8.14: Output after starting the daemon

And if we do an **lsof -i -P** to list the Internet sockets and port numbers (not port names), we see all of them listening on the correct socket (Figure 8.15).

```
[root@localhost conf.d]# lsof -i -P | grep 10000
php  3000  apache  16u  IPv4 9591   TCP *:10000 (LISTEN)
php  3001  apache  16u  IPv4 9591   TCP *:10000 (LISTEN)
php  3002  apache  16u  IPv4 9591   TCP *:10000 (LISTEN)
php  3003  apache  16u  IPv4 9591   TCP *:10000 (LISTEN)
php  3004  apache  16u  IPv4 9591   TCP *:10000 (LISTEN)
php  3005  apache  16u  IPv4 9591   TCP *:10000 (LISTEN)
php  3006  apache  16u  IPv4 9591   TCP *:10000 (LISTEN)
php  3007  apache  16u  IPv4 9591   TCP *:10000 (LISTEN)
php  3008  apache  16u  IPv4 9591   TCP *:10000 (LISTEN)
php  3009  apache  16u  IPv4 9591   TCP *:10000 (LISTEN)
```

Figure 8.15: All the forked processes listening on the same socket

Does it run? I placed the PHP manual on my server so that I could test indexing a decent-sized volume of information. If I go to the form I created earlier, type the URL where the documentation is available, set the "recursive" checkbox to true, and submit the page, the Web page immediately returns with the "URL Submitted" message. The daemon is quite different.

Figure 8.16 shows the output generated after submitting the URL to the daemon. (Due to page width limitations, this version of the output includes a couple of line breaks for readability.)

```
Child 3008 started...
Child 3009 started...
PID 3000 received URL http://dev/php/html/index.html
PID 3001 received URL http://dev/php/html/preface.html
PID 3002 received URL http://dev/php/html/copyright.html
PID 3003 received URL
 http://dev/php/html/getting-started.html
PID 3004 received URL
 http://dev/php/html/introduction.html
PID 3004 received URL http://dev/php/html/tutorial.html
PID 3005 received URL http://dev/php/html/manual.html
PID 3004 received URL http://dev/php/html/install.html
. . .
```

Figure 8.16: Output after submitting a URL

The daemon immediately starts processing the pages, using up much of my free CPU very quickly. Because we index each document, the bottleneck is the writing to the index. But even so, I am processing multiple pages at a time, doing a fair number each second. Once the daemon has parsed and indexed all the pages, it generates the output shown in Figure 8.17 (again with line breaks introduced).

```
PID 3030 received URL
 http://dev/php/html/function.imagickdraw-bezier.html
PID 3029 received URL
 http://dev/php/html/function.imagickdraw-arc.html
PID 3026 received URL
 http://dev/php/html/function.imagickdraw-annotation.html
PID 3024 received URL
 http://dev/php/html/function.imagickdraw-affine.html
Data received. Optimizing index...
Optimizing Complete.
```

Figure 8.17: Output after site has been spidered

At this point, we can leave the daemon running to handle more connections or press **Ctrl-C** to get out of it. All that's left to do is to test the search on the

machine. When I type in **pcntl** as my search query, I get the output shown in Figure 8.18.

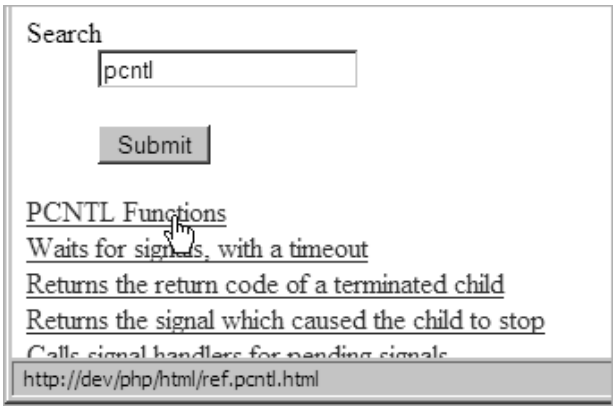

Figure 8.18: The search results

Inter-Process Communication

We could leave it at that, but there is one more thing we might want to look at, primarily because it's kind of a neat way of distributing functionality. What I would like to do is demonstrate how, by using a combination of objects and serialization, you can actually build a very compelling distributed system that can be both dumb and smart at the same time.

Let's start by defining an interface. This interface will be used by an individual worker to discover whether the received object was a **Url** object that we had previously defined or an object that contains executable functionality. We can use this interface as a sort of switch to help make that determination. Figure 8.19 shows the interface definition.

```
interface CmdInterface
{
      public function execute();
}
```

Figure 8.19: Interface for any executable functionality

Simple enough. For our example, the command that we want to build here is a command that forks another process based on the answering process upon request (Figure 8.20).

```
class CmdFork implements CmdInterface
{
      public function execute()
      {
            if (($pid = pcntl_fork()) > 0) {
                  echo "New child {$pid} started...\n";
            }
      }
}
```

Figure 8.20: Class designed to fork a child

The class basically has one line of functional code. However, based on whatever functionality you need, it could have as much code as you want and as much data as you need.

We need a simple user interface (UI) to kick this off (Figure 8.21).

```
if (isset($_GET['fork'])) {
      $fork = new CmdFork();
      $serverSock = fsockopen('tcp://dev', 10000);
      $data = serialize($fork);
      $data = pack('N', strlen($data)) . $data;
      fwrite($serverSock, $data);
      fflush($serverSock);
      echo 'Request has been sent';
} else {
      ?>
Would you like to have another forked worker?
<a href="?fork=y">Yes</a>
      <?php
}
?>
```

Figure 8.21: Sending the fork command to the daemon

To implement the command, we create a new object of **CmdFork**, serialize it, and send it to the daemon. That's easy enough. The only thing left to do now is make

our daemon aware of it. For that, we'll modify the **_execute()** method. Figure 8.22 highlights the necessary change.

```
private function _execute()
{
      while (true) {
            $sock = socket_accept($this->_sSock);

            $obj = $this->_readData($sock);
            if ($obj instanceof CmdInterface) {
                  $obj->execute();
                  socket_close($sock);
                  continue;
            } else if (!$obj instanceof Url) {
                  socket_close($sock);
                  continue;
            }

. . .

      }
}
```

Figure 8.22: Modifying the _execute() call to recognize CmdInterface

That is all we need to do. We start our daemon back up, click on the forking link on our UI, and send the command to the daemon. Figure 8.23 shows our output.

```
[apache@localhost Daemon]#
  /usr/local/zend/bin/php daemon.php
Child 2931 started...
Child 2932 started...
Child 2933 started...
Child 2934 started...
Child 2935 started...
Child 2936 started...
Child 2937 started...
Child 2938 started...
Child 2939 started...
Child 2940 started...
( . . . and then later on)
New child 2943 started...
```

Figure 8.23: Output after sending fork request

After I submitted the Web page, which in turn submitted the **CmdFork** object, a new process with the ID of 2943 was started. Let's take a look at our network connections (Figure 8.24) and see if this process is ready to go.

```
[root@localhost ~]# lsof -i -P | grep 10000
php  2931  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
php  2932  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
php  2933  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
php  2934  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
php  2935  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
php  2936  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
php  2937  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
php  2938  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
php  2939  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
php  2940  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
php  2943  apache  16u  IPv4 10891  TCP *:10000 (LISTEN)
```

Figure 8.24: Output after fork request

Yep. Process 2943 is sitting on the same socket as all the others, ready to handle the request.

Conclusion

The idea that is exciting here is that with this type of functionality, you can actually run PHP code as a background process by using a relatively dumb daemon. In fact, with our spidering example, we could even have written the code as a command, removing much of the functionality from the daemon itself. With that approach, the daemon would become much more multipurpose while still being able to do our spidering activity via a command instead of directly in the daemon. From a teaching perspective, I believe that the order of the ideas presented here is a better approach. But now that you have seen both possibilities, let your imagination run wild with the things you might be able to do.

Now, let's go back to my note at the beginning of the chapter. Having seen the code in this chapter, you might think this is something that a lot of organizations should be doing. I would argue that that is not the case. In most situations, the traditional PHP Web HTTP request/response approach is the proper way to write your application. Don't try to be too creative with your solutions. Be practical. Also, if you have an application that requires very high performance and a large back-end

infrastructure, this might not be the approach that you want to take either. PHP running as a daemon does not have the type of ecosystem that a language such as Java or C has.

However, the vast majority of Web sites out there do not require a massive back-end infrastructure, and many of them do have needs for some kind of asynchronous processing. If you are working on one of those sites, then something like what we've covered in this chapter might be pertinent to what you are doing.