

Server-side Caching Optimisations for the Apache Web Server

Simon Geard

June 6, 2000

Contents

1	Server Side Scripting	5
1.1	Server-side scripting	5
1.2	Scripting tools	5
1.2.1	PHP Hypertext Preprocessor	5
1.2.2	Active Server Pages	6
1.2.3	Java Server Pages	6
1.2.4	Server Side Includes	7
1.2.5	Comparisons	7
1.3	SHTML	7
1.3.1	SHTML Examples	8
2	Caching SHTML	10
2.1	Why is caching needed?	10
2.2	Rules	10
2.3	The caching process	11
2.4	Storing caching data	12
2.5	Variables	13
2.6	Sub-requests	13
3	Implementation	16
3.1	Apache version	16
3.2	Isolated processes	16
3.2.1	File locking	17
3.3	Coding	17
4	Further extensions	20
4.1	Client-side caching	20
A	Sample SHTML collection	22
A.1	foo.shtml	22
A.2	foo-body.html	22
A.3	bar.shtml	22
A.4	bar-body.html	23
A.5	header.shtml	23

A.6	footer.shtml	23
B	Information sources	24
B.1	PHP: Hypertext Preprocessor	24
B.2	Active Server Pages	24
B.3	Java Server Pages	24
B.4	Server-Side Includes	25

List of Figures

2.1	The caching process	11
2.2	Example dependency file: foo.shtml\$dep	12
3.1	Modifications to the entry function send_parsed_file().	18
4.1	The caching process supporting client-side caching	21

List of Tables

2.1	Example of the Apache environment table	15
-----	---	----

Chapter 1

Server Side Scripting

1.1 Server-side scripting

Most people are familiar with the idea of client-side scripting. Here, Javascript or other programming commands are embedded in an HTML document, and processed by the client on receipt of the document.

Server-side scripts work in basically the same manner, except that they are processed by the web server. When a scripted document is requested, the server parses the scripts, and returns the resulting document.

From the user point of view, a scripted document cannot be distinguished from an unscripted document. If they view the source returned by the server, all they will see is normal HTML - all server-side scripts are removed in the parsing process. To the user, the only hint may be that the file has a different extension - .php or .asp instead of .html.

1.2 Scripting tools

There is a wide selection of scripting tools currently available. Listed below are the four that are most frequently encountered.

1.2.1 PHP Hypertext Preprocessor

The PHP Hypertext Preprocessor (PHP) is a server-side scripting system originally developed as a module for the Apache web server, running under UNIX environments. It can, however, be built as an executable to run under other UNIX servers, and binaries can be downloaded to use PHP on a Windows platform, such as Microsoft's Internet Information Server.

The PHP language is close to C++ or Java in syntax, although considerably simplified. It is object-oriented, but objects are not as integral as they are in Java.

An example of a site built with PHP is the Electronic Campus Webmail service, at <http://webmail.ec.auckland.ac.nz>. Here, PHP is used to provide a web-based interface to the student email system.

1.2.2 Active Server Pages

Active Server Pages (ASP) is Microsoft's server-side scripting language. Microsoft provide ASP for both their servers - Internet Information Server, and Personal Web Server, but other companies (most notably, Chillisoft) have implemented ASP for other web servers and operating systems.

Like most of the scripting tools provided by Microsoft, ASP is based on the BASIC language, and more specifically, Microsoft's VisualBASIC. As such, anyone familiar with Windows development should have no difficulty understanding ASP.

An example of ASP being used in a page is <http://www.aspforums.com>. ASP Forums produce a variety of ASP-based web applications, including authentication systems, discussion forums, and a link category service similar to Yahoo.

1.2.3 Java Server Pages

Java Server Pages (JSP) is the Java solution, designed by Sun Microsystems, and part of Sun's Servlet technology. Implementations of both are available for all major web servers - the Apache Group has several projects based on Java technology.

The key difference between JSP and other server-side scripting technologies is that JSP is compiled. Instead of simply parsing a text file, the Servlet engine compiles the JSP to a servlet the first time it is used. For further requests, the servlet is used, without reference to the JSP.

There are very few sites using JSP at present. Merge Technologies, at <http://www.merge.com> is using JSP throughout its site, but it doesn't do anything that couldn't be done with a simpler system like SSI.

1.2.4 Server Side Includes

Server-Side Includes (SSI) is a server-side scripting system provided by Apache in the form of the add-on module *mod_include*. It's not exclusive to Apache, but that is the platform it is most often used on.

SSI is a very different system to the others, in that it does not provide a full programming language. Instead, it provides a series of directives which the parser replaces with the requested information, such as the value of an environment variable, or the contents of another file.

An example of SSI being used is Compuware Numega, whose site is at <http://www.numega.com>. Numega have used SSI documents for their entire site, though it's impossible to tell from the client side what kind of features they use.

Note: though SSI is the correct name for this system, it is usually referred to as SHTML (Server-parsed HTML), probably because this is the standard file extension used by SSI documents. In any case, this is the term I use throughout this report.

1.2.5 Comparisons

PHP, ASP and JSP all offer basically the same functionality. They're complete programming languages, with the same kind of abilities as C, Java, or Basic. All of them have extensive database support, and some ability to control the server environment.

The main difference is the vendors. ASP is the Microsoft choice, and is at its best running on Windows servers. PHP, in contrast, is a product of the Open Source movement, and is used extensively by Open Source websites. JSP is the alternative introduced by Sun, who are linking Java into all their projects. JSP is the newest of the three, and does not yet have the user base that the others do.

1.3 SHTML

As stated above, SHTML is an extremely simple system. Unlike the other systems listed above, SHTML is not intended for providing dynamic content, such as a database application. Rather, it is well suited to reducing the maintenance required for a site, by embedding other files, and displaying simple information about files.

SHTML consists of a series of directives embedded in HTML comments. The table below is a complete list of directives supported.

config changes the appearance of certain types of output - the format of file size, and the format used for printing date/time values.

echo displays the value of an environment variable.

exec executes an external command.

fsize displays the size of a specified file.

flastmod displays the last-modified timestamp of a specified file.

include inserts the contents of another file into this document. The included file is handled by Apache, and thus may be of any type that produces text output - plain text, HTML, or even another SHTML file.

printenv displays a list of all defined variables and their values.

set assigns a value to a variable.

if/elif/else/endif provide conditional display of sections of an SHTML document.

1.3.1 SHTML Examples

Appendix A contains a number of SHTML and HTML documents which are used as examples throughout this report. The function of these is outlined here, as an example of what can be done with SHTML.

Foo.shtml is a top-level document, which may be requested by a client. It contains a basic HTML structure, plus four SHTML directives:

```
<!--#set var="LOCATION" value="foo" -->
<!--#include virtual="header.shtml" -->
<!--#include virtual="foo-body.html" -->
<!--#include virtual="footer.shtml" -->
```

The first line creates an environment variable `LOCATION`, and assigns it the value 'foo'. This variable is visible in this file, and in any files that this file *includes*. The next three lines import other documents. In each case, a sub-request is made to Apache, and the output is sent to the client as part of foo.shtml.

Header.shtml demonstrates the conditional-display directives to display a simple menu.

```
<!--#if expr="\$LOCATION\" = \"foo\" -->
  [ Foo ]
  [ <a href="bar.shtml">Bar</a> ]
<!--#elif expr="\$LOCATION\" = \"bar\" -->
  [ <a href="foo.shtml">Foo</a> ]
  [ Bar ]
<!--#endif -->
```

Basically, it identifies the parent document by examining the LOCATION variable created in the parent. If this value is set to ‘foo’, then the link to foo.shtml is disabled. If set to ‘bar’, then the link to foo.shtml is enabled, and the link to bar.shtml is disabled instead. This is a simple way of providing feedback on the menu without making a site difficult to maintain.

Footer.shtml is a simple piece of code for showing the date the requested file was last modified.

```
Last modified: <!--#echo var="LAST\_MODIFIED" -->
```

Nice and simple, the echo directive prints out the value of the specified variable. LAST_MODIFIED is a variable defined by Apache, and contains the date/time that the requested file was last modified. Note, this does not display the timestamp of footer.shtml, but that of the document requested by the user. In this case, foo.shtml.

Foo-body.html is a normal HTML file. When it is requested by the parent, Apache simply delivers it as is, without parsing it. For documents that do not contain SHTML directives, it is better to keep them as unscripted HTML. For the standard module, this helps reduce load on the server; for the caching version developed for this project, it also reduces the amount of data that needs to be cached.

Chapter 2

Caching SHTML

2.1 Why is caching needed?

Under the existing implementation of SHTML, documents are regenerated every time a client makes a request. While SHTML is a simple language to deal with, re-parsing a document when not necessary may not be the most efficient means of operating a web server.

In theory, it should be possible to increase the efficiency of the SHTML module by implementing a caching mechanism. Because SHTML is a mostly static system, it is possible to create a simple set of rules to determine if a document can be cached, and a dependency-checking system to determine if a cached copy is valid.

2.2 Rules

The list below describes the SHTML directives that will have an effect on caching of a file.

exec - Executes an external program or script. Because the output of such a program cannot be determined until after it has run, caching is not possible for any document that makes use of this directive.

printenv - Displays the current environment. Because the environment includes information like the current date/time, a document containing this directive cannot be cached.

fsize, **lastmod** - Display the size and last-modified timestamp of a specified file. This introduces a dependency on the given file.

include - Reads in a specified file, and inserts it at the point of the directive. This introduces a dependency on the given file. If the file is another

SHTML document, then it may introduce its own dependencies when parsed.

In addition, any directive may introduce a dependency on an environment variable. In `footer.shtml` for example, an *echo* directive refers to the value of the `LAST_MODIFIED` variable. Similarly, `header.shtml` refers to a variable `LOCATION`, which is defined by either `foo.shtml` or `bar.shtml`.

2.3 The caching process

Figure 2.1 shows the process involved in delivering a document from a server cache. If the request is in the cache and the cached copy is still valid, then the cached copy can be returned to the server. Otherwise, the document must be re-parsed first, and the cache updated.

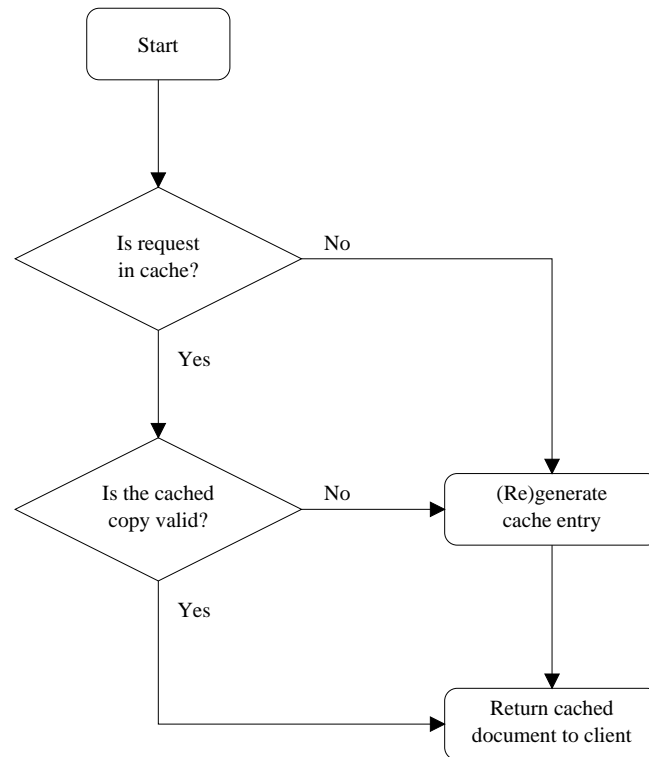


Figure 2.1: The caching process

2.4 Storing caching data

The Apache environment is a dynamic one, where processes may be created or destroyed at any time. Because of this, it is difficult to make efficient use of shared memory.

Instead, a purely disk-based system will be used, with requested documents corresponding to files in a cache directory. This system ensures persistence of the caching data, and is a simple solution to the problem of sharing data between processes.

Two types of file will exist. For each SHTML file requested by the client, a dependency file will exist, e.g `foo.shtml:dep`. This file will contain a table of all the dependencies associated with the requested file, and for any SHTML files *included* as part of this file.

The second file type contains the cached output of an SHTML file. Cache files will exist for every SHTML file that passes through the module, and quite possibly more than one. For *included* files, a cache may exist for each parent that refers to them. For example, `header.shtml` would have two cache files - `foo.shtml:header.shtml:cache`, and `bar.shtml:header.shtml:cache`.

Cache-table entries

Figure 2.2 shows an example of what the cache entries might look like for the file `foo.shtml`. A *\$cache* extension refers to the cached data, thus the cache entry for `/foo.shtml$cache` indicates dependencies on two normal files (`foo.shtml` and `foo-body.shtml`), and dependencies on two SHTML files, which have their own sub-entries.

<code>foo.shtml\$cache :</code>	<code>foo.shtml</code> <code>foo.shtml:header.shtml\$cache</code> <code>foo-body.shtml</code> <code>foo.shtml:footer.shtml\$cache</code>
<code>foo.shtml:header.shtml\$cache :</code>	<code>foo.shtml</code> <code>header.shtml</code>
<code>foo.shtml:footer.shtml\$cache :</code>	<code>foo.shtml</code> <code>footer.shtml</code>

Figure 2.2: Example dependency file: `foo.shtml$dep`

2.5 Variables

Two kinds of variable may be identified for the caching system, and these must be handled in two separate ways.

Some variables are defined within SHTML files. For example, both `foo.shtml` and `bar.shtml` use the *set* directive to define a variable called `LOCATION`. Tracking this value can be handled simply - by recording a dependency on the file in which `LOCATION` is defined.

This is why a file like `header.shtml` may have multiple cached copies. One copy would reflect the dependency on `foo.shtml`, in which `LOCATION` is defined as `'foo'`. A second copy would be dependent on `bar.shtml`, where the value is set as `'bar'`. In this way, file dependencies may be used to resolve variable dependencies.

The second type of variable is those defined by Apache in the environment table it passes to *mod_include*. Since these aren't defined in SHTML, they cannot be resolved in the same way as locally defined variables. Table 2.1 shows a selection of these variables, provided by the *printenv* directive.

The intended solution is to handle these case-by-case. `LAST_MODIFIED` and `DOCUMENT_URI` can be handled simply by recording a dependency on the appropriate file. Others, such as `DATE_LOCAL` or `REMOTE_ADDR` cannot usefully be cached anyway, and so will cause the document to be marked uncacheable. Others, such as the server properties might be cacheable, but will not be handled at first. Later development may find a way to cache documents that refer to these variables.

2.6 Sub-requests

Sub-requests will be cached specifically for the parent document, to avoid conflict when a file is *included* by different parents. Thus, cache entries will have a qualified name. Since `foo.shtml` and `bar.shtml` both *include* the files `header.shtml` and `footer.shtml`, there will be six entries in the cache, as below:

- `foo.shtml`
- `foo.shtml:header.shtml`
- `foo.shtml:footer.shtml`

- `bar.shtml`
- `bar.shtml:header.shtml`
- `bar.shtml:footer.shtml`

SHTML files only

Only SHTML files will have their own cache entries. If they *include* other file types, the content of those files will be stored as part of the parent, but they will not have cache entries of their own.

In the previous example, note that the included files `foo-body.html` and `bar-body.html` do not have cache entries. Their contents will be included in the cache entry for `foo.shtml` and `bar.shtml`, but since they don't need parsing, nothing will be gained by caching them, other than to create duplicate files.

DOCUMENT_ROOT	/home/httpd/htdocs
HTTP_ACCEPT	*/*
HTTP_ACCEPT_CHARSET	iso-8859-1,*,utf-8
HTTP_ACCEPT_ENCODING	gzip
HTTP_ACCEPT_LANGUAGE	en
HTTP_CONNECTION	Keep-Alive
HTTP_HOST	localhost
HTTP_USER_AGENT	Mozilla/4.7 [en] (X11; I; Linux 2.2.5-15 i686)
PATH	/sbin:/usr/sbin:/bin:/usr/bin:/usr/X11R6/bin
REMOTE_ADDR	127.0.0.1
REMOTE_PORT	1161
SCRIPT_FILENAME	/home/httpd/htdocs/test.shtml
SERVER_ADDR	127.0.0.1
SERVER_ADMIN	delgarde@geocities.com
SERVER_NAME	delgarde.home
SERVER_PORT	80
SERVER_SOFTWARE	Apache/1.3.12 (Unix) PHP/3.0.16
UNIQUE_ID	OTtM3cCoAAwAAAJHBHQ
GATEWAY_INTERFACE	CGI/1.1
SERVER_PROTOCOL	HTTP/1.0
REQUEST_METHOD	GET
QUERY_STRING	
REQUEST_URI	/test.shtml
SCRIPT_NAME	/test.shtml
DATE_LOCAL	Monday, 05-Jun-2000 18:46:53 NZST
DATE_GMT	Monday, 05-Jun-2000 06:46:53 GMT
LAST_MODIFIED	Thursday, 01-Jun-2000 22:07:11 NZST
DOCUMENT_URI	/test.shtml
DOCUMENT_PATH_INFO	
USER_NAME	root
DOCUMENT_NAME	test.shtml

Table 2.1: Example of the Apache environment table

Chapter 3

Implementation

The previous chapter outlined the mechanism by which the caching system will operate. This chapter goes into greater detail about how the mechanism will be implemented, and the limitations imposed by the way Apache and the existing SHTML module are structured.

3.1 Apache version

There are currently two versions of the Apache server. The first is the 1.3.x family, which has been around for a long time, and for now, has a much larger user base. The second is the newer 2.0.x family, which has major changes to the workings of the server, and some minor changes to the module system.

As a result of these changes, the SHTML module from 2.0.x has some differences from the version distributed with 1.3.x. Many are cosmetic changes such as renaming a function, but some parts of the code appear to have been revised, possibly because of the Apache changes.

Because of this, there is the need to create two versions of the module. Implementation will start with the 1.3.x version, since this is the more widely used version of Apache. Once this is completed and tested, reimplementing the changes on the 2.0.x version should not be difficult.

3.2 Isolated processes

In the Apache 1.3.x environment, each process handles a single client connection at a time. While 2.0.x changes this by adding multi-threading support, the problem of sharing resources between processes remains.

3.2.1 File locking

By keeping all caching data on disk, this problem can be solved relatively easily. POSIX compliant operating systems provide functions for file locking - these can be used to coordinate shared access to files.

Every time a process reads from a cache entry, it should obtain a shared lock for that file. If the cache entry is invalid and a process needs to update it, that process can obtain an exclusive lock, forcing other processes to wait before accessing it themselves.

3.3 Coding

The entry point to *mod_include* is the function `send_parsed_file()`. Apache passes this function a request structure. At present, this function looks at file permissions and Apache configuration, determining whether or not *mod_include* can handle the file. If so, it sets up the environment, and calls `send_parsed_content()` to do the work of parsing the file.

This function is the ideal place for the caching system to be added. Once the function has agreed to handle the document, it should then lookup the request in the caching table and check the dependencies. If they match, the cached file can be loaded and sent; otherwise, the normal (re)parsing process must occur.

Figure 3.1 shows how this part of the modifications will fit in with the existing system. Basically, it acts as a conditional statement. If the cached copy is valid, skip over the part that parses the file. Otherwise, it will run through it, parsing the file and adding/updating the cache.

This is only the part that determines whether or not the cached copy is valid. All functions that perform output must be modified to write to the cache file, instead of sending directly to the client. And the functions that handle directives must be modified to record the dependencies of the document.

There are two functions which must invalidate caching for a document which calls them. Both are the handlers for directives which will produce either undeterminable, or constantly changing results.

`handle_printenv()` handles the *printenv* directive. Because the output of this function includes values like the current system time, the simplest thing to do is to not cache documents that use it.

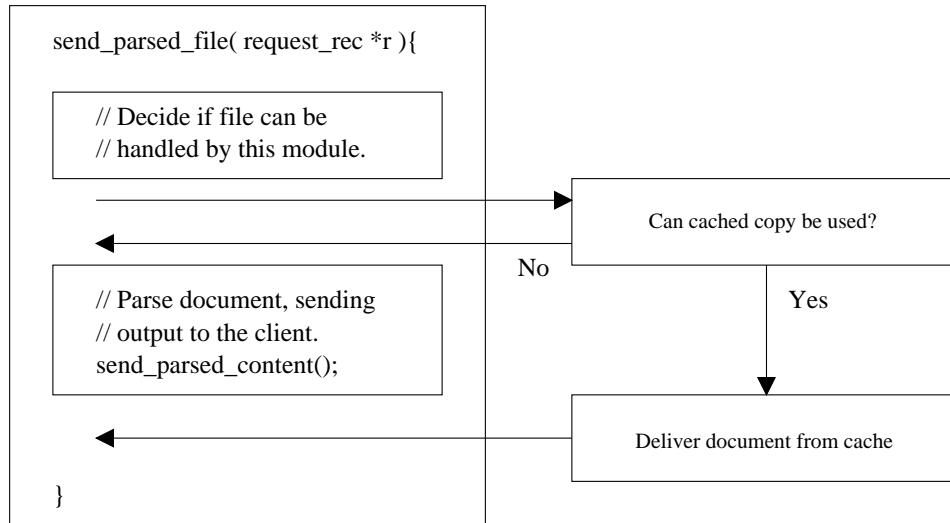


Figure 3.1: Modifications to the entry function `send_parsed_file()`.

handle_exec() handles the *exec* directive. Since running external programs makes the document content undeterminable, there is no way such a document can be cached.

Three more directives make use of files; the functions that implement these directives must be changed to record the dependencies.

handle_fsize() handles the *fsize* directive. This creates a dependency on the specified file, so the function must be modified to record the dependency.

handle_lastmod() handles the *lastmod* directive. The behaviour is exactly the same as for *fsize*.

handle_include() handles the *include* directive. Once again, this function must handle the file dependency that results.

Finally, all directives may contain references to variables, either defined in the document, or provided by Apache as part of the system environment. Most of the handlers call the *parse_string()* function to perform variable substitution, but a few perform the operation themselves. Thus, the following are the functions that need to be changed in order to track variables.

parse() is a general purpose function, accepting a string parameter, and expanding the value of any variables it encounters. It's called by the handlers of all the directives except *echo*.

handle_echo() is the handler for the *echo* directive. Since the only parameter it accepts is the name of a variable, it isn't necessary to call a separate function to do the job.

Chapter 4

Further extensions

While designing the server side caching system, it turned out that a further extension might be possible, extending the server-side caching mechanism to allow client-side caching.

4.1 Client-side caching

As defined by the HTTP protocol, client-side caching is provided by a directive If-Modified-Since. If the requested resource has not changed since the given time, the server will report this, and will not send the data.

Under the standard SHTML implementation, this mechanism cannot be used, since there is no way to determine whether or not the document has been changed. Since a file may have dependencies on other files, the normal method of checking the files timestamp will not work.

However, in the process of developing server-side caching, a mechanism is being created to determine whether an SHTML document has been changed. Given this, it should not be difficult to add the extra functions to support client-side caching of these documents.

Figure 4.1 shows the modified caching process. Everything remains the same, up to the point where the cached data is returned to the client. Where the original would simply return the cached data, the modified version would run a third test, comparing the Is-Last-Modified header sent by the client with the timestamp of the cache entry. If the date sent by the client is older than the cache entry, the cached data should be sent. Otherwise, an HTTP 304 response should be sent, indicating that no data need be sent.

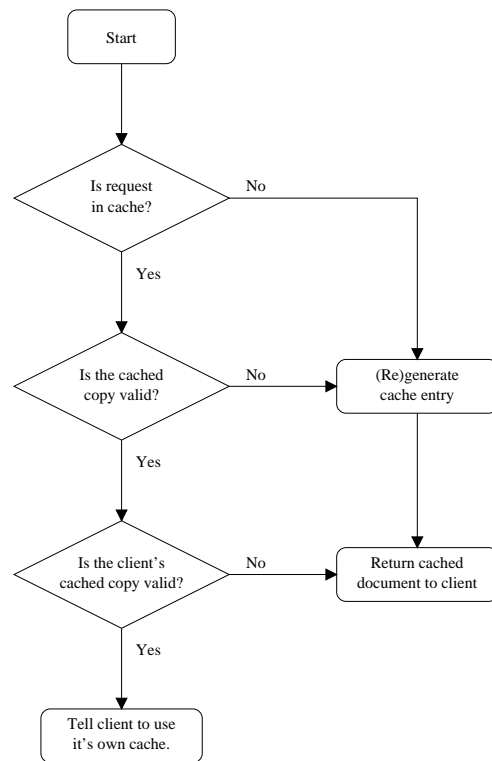


Figure 4.1: The caching process supporting client-side caching

Appendix A

Sample SHTML collection

This appendix contains SHTML and HTML documents used as examples throughout this report.

A.1 foo.shtml

```
<html>
<body>
  <h1>Welcome to Foo</h1>
  <!--#set var="LOCATION" value="foo" -->
  <!--#include virtual="header.shtml" -->
  <!--#include virtual="foo-body.html" -->
  <!--#include virtual="footer.shtml" -->
</body>
</html>
```

A.2 foo-body.html

```
<p>This is the content from foo_body.html</p>
```

A.3 bar.shtml

```
<html>
<body>
  <h1>Welcome to Bar</h1>
  <!--#set var="LOCATION" value="bar" -->
  <!--#include virtual="header.shtml" -->
  <!--#include virtual="bar-body.html" -->
  <!--#include virtual="footer.shtml" -->
</body>
</html>
```

A.4 bar-body.html

```
<p>This is the content from bar_body.html</p>
```

A.5 header.shtml

```
<div>
<!--#if expr="\$LOCATION\" = \"foo\" -->
[ Foo ]
[ <a href="bar.shtml">Bar</a> ]
<!--#elif expr="\$LOCATION\" = \"bar\" -->
[ <a href="foo.shtml">Foo</a> ]
[ Bar ]
<!--#endif -->
</div>
```

A.6 footer.shtml

```
<hr>
Last modified: <!--#echo var="LAST_MODIFIED" -->
```


Appendix B

Information sources

This appendix contains a list of sources which may provide further information about topics this report covers.

B.1 PHP: Hypertext Preprocessor

All the information you could want about PHP is available at the PHP home page.

<http://www.php.net>

B.2 Active Server Pages

There is no official site for ASP, other than the pages Microsoft devotes to Internet Information Server and Personal Web Server. Try searching Microsoft's site for information.

<http://www.microsoft.com>

Chillisoft is a company that develops ASP implementations for non-Microsoft platforms, such as Apache on Linux or other Unix variants.

<http://www.chillisoft.com>

B.3 Java Server Pages

Since JSP isn't very widely used yet, there aren't too many useful sites. One good place to look for information is JavaSoft, the developers of the Java language.

<http://www.javasoft.com>.

The Jakarta project is one of the groups developing Java-based systems for Apache. The project is run by the Apache Foundation.
<http://jakarta.apache.org>.

B.4 Server-Side Includes

This project deals with Apaches SSI implementation, so the best place to look for is the Apache documentation. There isn't much, but there is one documentation page dealing with `mod_include`.
http://www.apache.org/docs/mod/mod_include.html.