Application Note APLX-LMW-0403:

Interfacing the Apache Web Server to APLX

Introduction

This document describes how you can use APLX in conjunction with Apache, the widely-used web-server software developed by the Apache Group (http://www.apache.org/).

It allows your APLX applications to serve data requests to web pages accessed using any standard browser (such as Microsoft Internet Explorer or Apple Safari). If your system is permanently connected to the Internet, this makes it possible to host fully-functional web sites which obtain all or part of their content from an application written in APL.

In the system described in this note, you do not have to be concerned at all with the details of internet protocols; Apache handles all of the interaction with the client web browser, and typically it also serves up any static web pages which do not require an interface to APL. When the remote client requests a page (or submits a form) which requires data from the APL application, Apache will pass the request to your APLX application, and wait for the response. Your APLX application returns the data, and Apache then transmits the HTML page to the browser.

What is Apache?

Apache is an open-source web-server package which was developed by the Apache Group. It is an entirely volunteer effort, completely funded by its members, not by commercial sales. As a result, Apache is available free of charge. Apache is available for a very wide range of systems, including Linux, AIX, Windows and MacOS X.

Although Apache costs nothing to use, that does not in any way imply that it is an inferior product. On the contrary, a very large proportion of web-sites including many high-volume commercial sites are based on it. Many commercial Internet Service Providers use Apache to host web pages for their clients. As at February 2000, over *six million* sites were based on Apache; it accounts for well over half of all sites on the World Wide Web.

Advantages of using Apache

It is possible to write your own home-brew web server in APL, directly receiving HTTP requests over a TCP/IP network, and returning the results directly over the network. Although this approach can be made to work, it involves a non-trivial development effort. In

any case, it is better to take advantage of the Apache software to handle as much of the work as possible, for the following reasons:

- Apache is very reliable. It is well-tested, and is used by many millions of web-sites around the world. This is very important given that web sites may be accessed 24 hours a day, 365 days a year, from all around the world.

- It is multi-threaded, and provides a high-performance solution which can handle many simultaneous requests very efficiently.

- It is very easy to set up and use, and very easy to interface to APL.

- It provides many extra features such as error logging, and access logging (to keep track of who visits your site and the pages they access). Because of Apache's popularity, there are many packages available to help you interpret the web logs.

- Apache also handles more advanced features such as server-side includes, cookies, secure connections, content negotiation, redirection, and access control. Even if you do not need these features at present, you may want to use some of them in the future.

- Apache has been thoroughly checked for security holes, and has been designed to be robust in the face of denial-of-service attacks and other malicious attempts to disrupt your web site. It is also robust against accidental runaway conditions which could prevent your web-site working properly. It would be very hard to replicate this robustness in your own APL web server.

In summary, using Apache rather than serving up pages directly from APL provides a much better solution, with much less effort, at zero cost. Not a difficult decision!

Step 1: Configure Apache and setup some static web pages

If you want to try this approach of combining Apache and APL to serve up web pages, the very first thing to do is to start Apache running, and get it to serve up some static HTML pages.

In this note, we will focus on interfacing Apache to APLX under Linux. However, the techniques presented can also be used for other operating systems; this is described briefly at the end of this document.

Apache is included on most Linux distributions, and may well be configured and already running on your Linux system. To find out if it is already running, type `ps -ea` at the Linux command prompt, and see if any of the tasks are running the `httpd` program. Otherwise you will have to configure it by hand. Consult the Apache documentation or any introductory book on Linux for more information on this first step.

HTML pages are just text files, placed in the 'html' or 'docs' directory of the Apache installation. (On our test installation, the directory is `/home/httpd/html/`, but your system may be configured differently). Apache come with its own test page `index.html`,

so if you have got it set up correctly on a machine on your network, you can just point your browser at it using the URL http://XXX/ , where XXX is either the name or IP address of the server running Apache. If all is well, your browser will display the Apache test page `index.html`.

To access the page from an arbitrary machine on the internet, you would of course have to use the full internet IP address or a fully-qualified domain name like http://www.mycompany.co.uk, where the domain name had been registered as pointing to your system - check with your ISP for more information.

In our examples, we'll assume the Linux server running Apache has the hostname 'penguin', so the Apache test page is on our intranet at: http://penguin

Once you have the Apache test page working, you can create your own HTML pages, either by writing HTML directly in a text editor, or preferably by using a web-page editor such as the excellent (and free) '1st Page' from Evrsoft Pty Ltd (http://www.evrsoft.com/). If you create a page called `mypage.html`, and place it in the Apache HTML pages directory, you can access it from your browser using an URL in the form http://penguin/mypage.html. If you place it in a sub-directory 'aboutme' of the Apache HTML directory, the URL would be something like http://penguin/aboutme/mypage.html (remember to use the correct host name rather than 'penguin' if you try these!)

Step 2: Serving up dynamic content from a program or script

Once you are comfortable with using Apache with static HTML pages, the next stage is to understand how Apache calls another program to pick up dynamic pages, which it then serves up to the remote browser.

The answer is very simple. There should be a directory called 'cgi-bin' in your Apache directory tree. (On our system it is `/home/httpd/cgi-bin/`). *Any program placed in that directory is assumed by Apache to be the name of a script or program which provides dynamic content.*

The program or script can be written in almost any language (often Perl is used, but many others are possible). All it has to do is write the HTML for the page to standard output. When the page is requested, Apache will invoke the program and pick up its output via a pipe. It will then serve the output to the remote browser.

The first line of the output has to identify the content type. For HTML pages, this is of the form:

```
Content-type: text/html
```

followed by a blank line.

The HTML itself then follows.

Before we start thinking about interfacing to APL, we can test this interface by writing a little shell script which simply displays the local time. We can write the script as follows:

```bash
#!/bin/bash
# Little test shell script for Apache

# Output HTTP header and opening HTML tags.
echo "Content-type: text/html"
echo ""
echo "<html>"
echo "<head>"

# Output title of page
echo "<title>My first dynamic page</title>"

echo "</head>"
echo "<body>"

# Output a heading.
echo "<h1>Time of day</h1>"

# Output the body text
echo "<p>The local time at Global HQ is "
date
echo "</p>"

# Finish off page
echo "</body>"
echo "</html>"
```
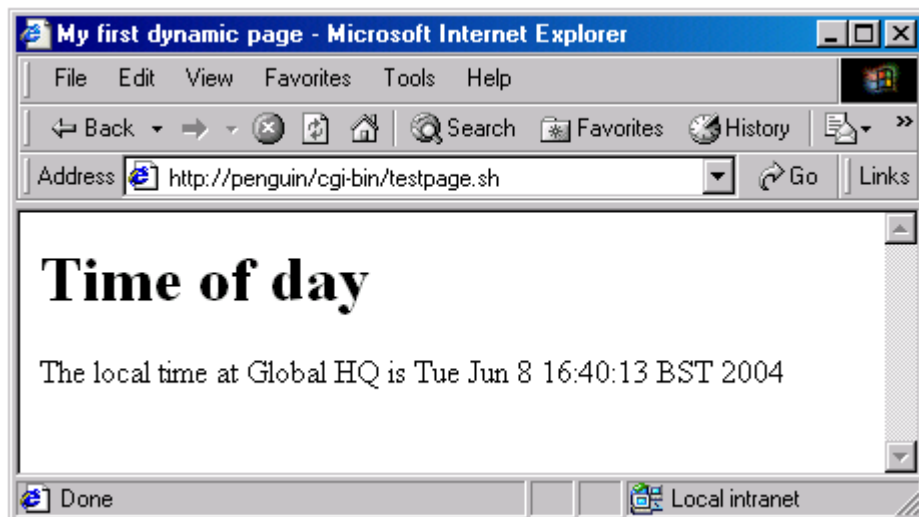
If we save this script as 'testpage.sh' in the cgi-bin directory (remembering to make it executable by anyone using chmod a+x testpage.sh), we can test it in isolation by just running it at the Linux command prompt:

```
$ cd /home/httpd/cgi-bin
$ ./testpage.sh
Content-type: text/html

<html>
<head>
<title>My first dynamic page</title>
</head>
<body>
<h1>Time of day</h1>
<p>The local time at Global HQ is
Tue Jun  8 16:37:05 BST 2004
</p>
</body>
</html>
```

It seems to do what we expected. Now let's see if Apache can run it and serve up the HTML to Internet Explorer. Any URL which refers to the 'cgi-bin' directory will be treated as a request to run a *program* in the cgi-bin directory, so in this case we want the URL to be
http://penguin/cgi-bin/testpage.sh

Internet Explorer displays the dynamic page based on the HTML which our little script outputs:

It really is that simple!

Note that Apache doesn't care what language the program is written in. It simply invokes the program or script, and picks up the text from standard output.

Step 3: Picking up parameters passed by Apache

In a realistic example, the dynamic content would usually depend on the exact query which the user submitted. You might also want to check other information about the request, such as the hostname of the machine making the request. Apache passes this information in a series of environment variables. There are quite a few of these, amongst which are the following:

`$REQUEST_METHOD`

> The method with which the request was made. This will be one of "GET", "HEAD", "POST", etc

`$QUERY_STRING`

> The query part of the request, with spaces converted to %20 (see below).

`$REMOTE_HOST`

> The name of the remote host making the request, if known.

`$REMOTE_ADDR`

> The IP address of the remote host making the request.

`$CONTENT_TYPE`

> For queries which have attached information, such as HTTP POST and PUT, this is the content type of the data.

`$CONTENT_LENGTH`

> The length of the content as given by the client.

In a shell script, all this data is of course available very simply as shell variables. In other languages, you need to check how to access environment variables from within the program.

To keep things simple, we'll continue our examples by typing the query directly in the URL line; the query part of the URL is anything after a question mark (you are probably familiar with this from the URL which appears in your browser location field when you use Google). Spaces get converted to %20.

For example, if the URL requested by the user was:

> http://www.mysite.com/myscript.sh?"What is the time?"

then `$QUERY_STRING` will be:

> "What%20is%20the%20time?"

(In a real application, you would typically use an HTML form to allow the user to request data; the HTTP request type would then be a POST.)

Step 4: Passing the data to APL and getting back the response

Now all we need to do is to figure out how to pass the query (which Apache gives us in `$QUERY_STRING`) to our APL application, and how to get APL to give us back the result when it is ready.

One obvious way of doing this is to invoke APL directly, either via a shell script, or replacing the whole shell script. For example, *APLX for Linux Server Edition* can get input from standard in, and send output to standard out. It could thus be used directly by Apache.

However, we don't really recommend this method, because starting up a whole new APL session on every request is too inefficient. It is better to have an APL server application sitting ready, waiting for a request to turn up. We just need to find a way to wake it up when we get a request, pass it some data, and get back the result. It can then get blocked so it doesn't use any CPU resource whilst it waits for the next request.

You can achieve this in a number of ways. We'll use a mechanism called 'named pipes'. These look and behave much like ordinary files, but are in fact 'First In, First Out' buffers which allow one process to write data which a second process reads. They are thus ideal for our purpose, and they are also quite efficient in Linux.

You create a named pipe using the 'mkfifo' program at the Linux command line. We'll create one in the /tmp directory, and make it accessible to anyone:

```
$ mkfifo /tmp/mypipe
$ chmod 666 /tmp/mypipe
```

If you want to see how named pipes work, you'll need to open a second Linux command window. In one of the windows, read from the pipe:

```
$ cat /tmp/mypipe
```

The process will sit there blocked, waiting for some data to come down the pipe.

In the other window, send it some data:

```
$ echo hello > /tmp/mypipe
```

The first process should now spring back into life and display 'hello'.

We can now make a shell script callaplx.sh which uses this mechanism to pass the Apache request to APL. We'll use a single pipe for queuing all requests, and a new pipe each time for APL to return the data to us. (Remember that there may be lots of requests arriving simultaneously; Apache will invoke our script in a new process for each one).

To prove the concept, we'll just pass the query part of the request to APL. Our shell script will add the header and HTML wrapper. Here's the complete, commented shell script:

```
#!/bin/bash

# callaplx.sh: Shell script so that Apache can invoke APLX

# Choose a name for the pipe via which APLX can return the result to us.
# We can use the process ID to make this unique for each invocation:
resultpipe=/tmp/resultpipe$PPID

# Create the pipe, and give it public access:
mkfifo $resultpipe
chmod a+w $resultpipe

# Write the query to the single named pipe we use for all input
# to the APLX server application,
# preceded by the pipe name so APLX knows where to return the result:
echo "$resultpipe|$QUERY_STRING" > /tmp/mypipe

# Read from the result pipe, blocking until APLX has something to
# return to us:
result=$(cat $resultpipe)

# Kill the result pipe now that we've got the result:
rm $resultpipe

# Output HTTP header and opening HTML tags:
echo "Content-type: text/html"
echo ""
echo "<html>"
echo "<head>"
```

```
# Output title of page
echo "<title>APLX-Apache Interface Demo</title>"

echo "</head>"
echo "<body>"

# Output a heading.
echo "<h1>Result of calling APLX</h1>"

# Output the query as sent by the browser
echo "<p>Query was: $QUERY_STRING</p>"

# Output the text APL returned to us
echo "<p>APLX returned: $result</p>"

# Finish off page
echo "</body>"
echo "</html>"
```

This uses `mkfifo` to create a new, unique named pipe through which the reply can be received. It uses the existing `/tmp/mypipe` as a queue to send the request (preceded by the name of the pipe where we want the result) to the APL application. It reads from the new pipe (and blocks until it is available), picks up the returned text, and outputs it in a pre-defined HTML wrapper. It also deletes the temporary pipe it has created.

On the APL side, we need to sit waiting for some data. The easiest way of doing this is to use ⎕HOST, as follows:

```
        ∇ProcessWebRequests;REQUEST;REPLY;DELIM;PIPE;⎕IO
[1]   ⍝ Wait for a request to come down the pipe, evaluate it using ⎕EA.
[2]   ⍝ The result is returned to the calling script via the return pipe
[3]   ⍝ The request is in the form RETURNPIPE|Request
[4]    ⎕IO←1
[5]   :Repeat
[6]      REQUEST←⎕HOST 'cat /tmp/mypipe'
[7]     :If 0≠⍴REQUEST
[8]       ⍝ Convert the %20 back to space, and extract the return pipe name
[9]       ⍝ Also convert $ to ⎕
[10]      REQUEST←⎕SS REQUEST '%20' ' '
[11]      :If REQUEST≡'bye' ◇ :Return ◇ :EndIf
[12]      DELIM←REQUEST⍳'|'
[13]      PIPE←(DELIM-1)↑REQUEST ◇ REQUEST←DELIM↓REQUEST
[14]      REQUEST←⎕SS REQUEST '$' '⎕'
[15]      'Got request to evaluate ',REQUEST
[16]      REPLY←("'Oops, cannot hack ',REQUEST") ⎕EA REQUEST
[17]      ⎕HOST 'echo "',(∊⍕REPLY),'" > ',PIPE
[18]     :EndIf
[19]   :EndRepeat
        ∇
```
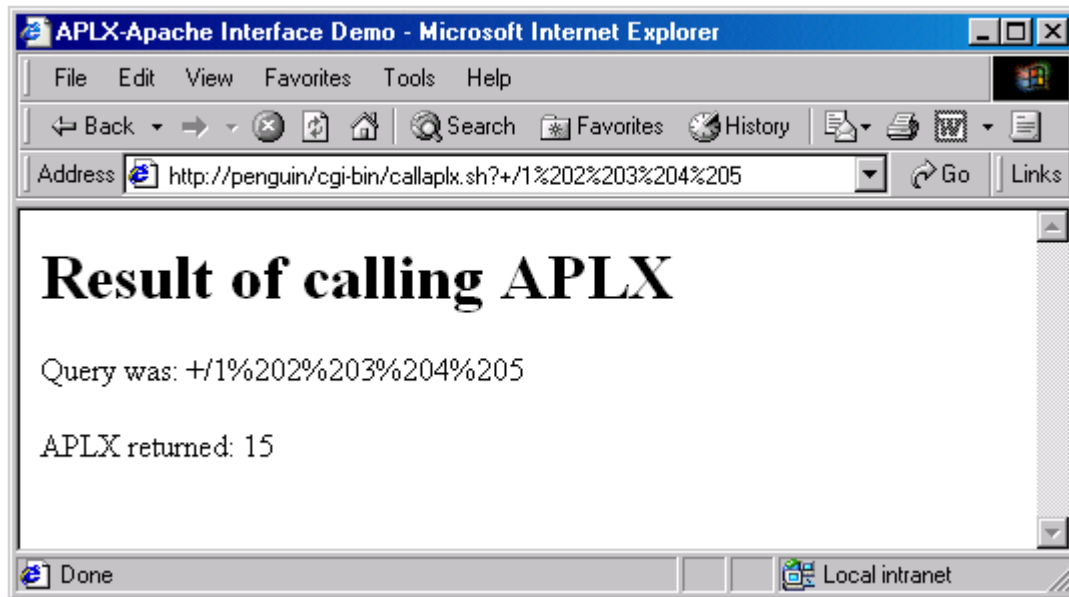
This program will loop round waiting for a request. When it gets one, it evaluates it using ⎕EA (having substituted ⎕ for $), and it returns the result via the pipe which the script created. It also writes the request to the session window as a debugging aid (line 15). If it receives the request 'bye', it exits (so you can terminate it by typing `echo bye > /tmp/mypipe` at the Linux prompt).

Let's try it. Start APLX on the Linux system, and run the above function. From the web browser, we'll pass a query which comprises a simple APL expression. In the browser, we type in the URL as:

http://penguin/cgi-bin/callaplx.sh?+/1 2 3 4 5

(The part up to the question mark is the location and name of the script; the part after the question mark is the query). Before transmitting this, Internet Explorer will convert spaces to %20. The effect is as follows:

The query looks unreadable because of the substitution of %20 for space, but APL correctly evaluated it as:

```
    +/1 2 3 4 5
```

Step 5: The Real World

Of course, the above example is not very realistic. You're unlikely to want to pass arbitrary APL expressions via the browser (and in any case there is a problem of the special APL characters). But you should now be able to see how this mechanism can be used to pass parameters from your web pages to APL, and get back the result. Here are some pointers to making this mechanism more robust and more general.

In our example, APL just returns a simple one-line string via the pipe. You might want it to return a whole web page, or at least a substantial part of the web page. The method shown (using `echo` to pass the text) is not suitable for long, multi-line text; instead, you could use the native file functions to create a temporary file containing the text, and just `cat` it to standard output in the script. In this mode, the return pipe would be used only as a synchronization method and to pass the name of the temporary file which APL should use.

Given the need for high reliability and 24-hour operation, you'll obviously need to add a lot of error checking and recovery code. It is a good idea to use several APL tasks. This has the advantage that if one takes a long time to complete a query, another query can be serviced at the same time. For example, your script could use the last digit of the process ID (`$PPID`) to select one of ten pipes to write to, and you could have ten APLX tasks handling requests.

In a real-world example, you'd also want some recovery mechanism so that if any of the APL tasks crashes or blocks for any reason, other APL tasks can continue to be used or the problem task can be automatically restarted.

In addition, you don't have to use the shell script and `⎕HOST` combination to implement the techniques in this paper. In fact a more robust implementation might be to use Perl or C to write the program which Apache invokes, and perhaps a custom-written Auxiliary Processor on the APL side. That would allow for better error checking, timeouts, and recovery. But the overall software architecture would remain much as described above.

Other Platforms

This note primarily focuses on the interfacing APLX to Apache under Linux. However, similar techniques can be used on other platforms supported by APLX, as follows:

MacOS X

Apache is distributed by Apple and is fully supported under MacOS X. The code samples in this note should work as shown, provided you have an up-to-date version of 'BSDSupport.bundle' (downloadable from the MicroAPL website) . You can create named pipes using the `mkfifo` program by bringing up a Unix-style console window using the Terminal program.

AIX

The examples shown above should work in exactly the same way as under Linux, with one minor change. Because the 'Bash' shell is not normally implemented under AIX, you need to change the first line of each of the shell scripts to read:

```
#!/bin/ksh
```

instead of:

```
#!/bin/bash
```

Windows:

Interfacing Apache to APLX under Windows NT/2000/XP can be done using the same principles as described above, but the details are somewhat different. Named Pipes are available, but you can't create and use them using simple console commands as you can in Unix-style systems. Instead, you have make Windows API calls such as `CreateNamedPipe`, which you can do using ⎕NA. In addition, you do not have the full power of the Unix-style shell scripting language under Windows.

Whilst a full discussion of using Named Pipes in Windows is outside the scope of this note, here are some pointers to get you started. Full information can be found in the Windows technical documentation.

Named Pipes are created using the `CreateNamedPipe` API call. This is very like creating a file; it takes the name of the pipe, together with information like the read/write mode and buffer sizes, and returns a handle to it. The name should be in the form '`\\.\pipe\XXX`', where XXX is the name you choose. Unlike in Linux and MacOS, the pipe is not persistent; it is deleted when all processes have closed the pipe.

The server task (in our case, the APL application) executes the Windows system call `ConnectNamedPipe`, which by default blocks until a client task (in our case, the program which Apache will invoke) connects to the pipe by trying to open it. Unfortunately, if the client task tries to open the pipe when the server task is not ready, it gets an error rather than waiting for it to be ready.

Once the connection is established, the client can write to the pipe using normal file operations, and the server (APL) can read from the pipe in the same way. The server calls `DisconnectNamedPipe` to close the connection, and can then call `CreateNamedPipe` again to wait for a new command.

Finally, the server closes the pipe using `CloseHandle`, and it is deleted.

APLX can interface to the pipe using ⎕NA to make the Windows calls. Without going into the details, here are some definitions we can use to define the necessary functions in APLX:

```
⎕NA 'U4 kernel32|CreateNamedPipeA <C[*] U4 U4 U4 U4 U4 U4 U4'
⎕NA 'U4 kernel32|ConnectNamedPipe U4 U4'
⎕NA 'U4 kernel32|ReadFile U4 >C[512] I4 >I4 U4'
⎕NA 'U4 kernel32|DisconnectNamedPipe U4'
⎕NA 'U4 kernel32|CloseHandle U4'
```

On the Apache side, for the purposes of demonstration we can use a batch file. Because Windows batch files are very limited, this is not a production-quality solution, but it illustrates the technique. We'll use a temporary file to get the result back from APL to the caller.

The APL server must be started first, because it creates the named pipe. It then waits for a client to send it a command. It writes the reply to a temporary file, and when this is created the caller writes it to standard output. This is the APLX side of the transaction:

```apl
      ∇ProcessWebRequestsWin;REQUEST;REPLY;DELIM;PIPE;TIE;ERR;X;⎕IO
[1]   ⍝ Wait for a request to come down the pipe, evaluate it using ⎕EA.
[2]   ⍝ The result is returned to the calling script (and hence Apache)
[3]   ⍝ via a temporary file
[4]   PIPE←CreateNamedPipeA '\\.\pipe\cmdpipe' 3 0 1 10000 10000 1000000 0
[5]   :If PIPE=¯1 ◇ ⎕ES 'Could not create the named pipe' ◇ :EndIf
[6]   ⎕IO←1
[7]   :Repeat
[8]      ⍝ Block, waiting for client to connect to the pipe with a request
[9]      ERR←ConnectNamedPipe PIPE,0
[10]     ⍝
[11]     ⍝ Read the request from the pipe. Returns (ERRFLAG) (DATA) (LENGTH)
[12]     X←ReadFile PIPE '' 512 0 0
[13]     ERR←DisconnectNamedPipe PIPE
[14]     :If 0=1⊃X ◇ ⎕ES 'Pipe read error' ◇ ERR←CloseHandle PIPE ◇ :EndIf
[15]     REQUEST←((3⊃X)↑2⊃X)~⎕R,⎕L,'"'
[16]     :If (3↑REQUEST)≡'bye' ◇ ERR←CloseHandle PIPE ◇ :Return ◇ :EndIf
[17]     ⍝
[18]     ⍝ Convert the strange %20 back to space and $ to ⎕
[19]     REQUEST←⎕SS(REQUEST;('%20';'$');(' ';'⎕'))
[20]     ⍝
[21]     ⍝ For debugging, display request to session window
[22]     'Got request to evaluate ',REQUEST
[23]     ⎕CC ⎕R,⎕L  ⍝ Force immediate flush, so it displays in sync
[24]     ⍝
[25]     REPLY←∈⍕("'Oops, cannot hack ',REQUEST")⎕EA REQUEST
[26]     ⍝
[27]     ⍝ Write reply as a temporary file, then rename it
[28]     TIE←'c:\temp\xx.tmp' ⎕NCREATE 0 ◇ REPLY ⎕NWRITE TIE ◇ ⎕NUNTIE TIE
[29]     'c:\temp\aplreply.txt' ⎕NRENAME 'c:\temp\xx.tmp'
[30]  :EndRepeat
[31]  ERR←CloseHandle PIPE
      ∇
```

The batch file looks like this:

```
@echo off
REM Windows batch file for sending requests to APLX via a named pipe

REM Write the query to the single named pipe we use for all input
REM to the APLX server application
del c:\temp\aplreply.txt 2>NUL:
echo "%QUERY_STRING%" > \\.\pipe\cmdpipe

REM Wait for APL to write out a temporary file containing the reply
:waitloop
if not exist c:\temp\aplreply.txt goto :waitloop

REM Output HTTP header and opening HTML tags.
echo Content-type: text/html
echo.
echo ^<html^>
echo ^<head^>

REM Output title of page
echo ^<title^>APLX-Apache Interface Demo^</title^>

echo ^</head^>
echo ^<body^>
```

```
REM Output a heading.
echo ^<h1^>Result of calling APLX^</h1^>

REM Output the body text
echo ^<p^>Query was: %QUERY_STRING%^</p^>

REM Output the text APL returns to us
echo ^<p^>APLX returned:
type c:\temp\aplreply.txt
echo ^</p^>

REM Finish off page
echo ^</body^>
echo ^</html^>
```

To provide a robust solution, this would need to be enhanced in a number of ways. The most important is to take account of multiple requests coming in simultaneously; if the APL application was already busy when a task tries to write to the pipe, an error would be generated which is not trapped in this simple example. Also you would want to use a different temporary file name for each request, so that there is no danger of a reply to request A being sent to client B! As under Linux, you would probably want to use multiple APLX tasks in order to be able to deal with multiple requests simultaneously.

Implementing such features requires a more powerful programming environment than Windows batch files; it can be done much more easily in C or Perl. This makes it possible to use the synchronization features of named pipes to block tasks with timeouts, providing an efficient but robust interface which can recover when things go wrong.