Paper 100-30

Extreme Web Access: What to Do When FILENAME URL Is Not Enough

Garth Helf, Hitachi Global Storage Technologies, San Jose, CA

ABSTRACT

This paper shows you how to write an automated Web browser in SAS®. One reason you might want to do this is to extract data from useful websites, so you can load it into a data warehouse. SAS provides the URL access method to read HTML from web pages, but we will see why it won't work for many websites. Instead, we will use the SOCKET access method to perform the basic functions of a web browser. We will understand how a web browser works by learning about Hypertext Transfer Protocol (HTTP) – the standard technique for sending HTML over the internet. We will see how to use an HTTP viewer program like httpExplorer to observe the HTTP request sent to a web server and the HTTP response sent back to the browser, so that we can replicate the same communication in a SAS program. Finally, we will see how to handle some special cases, like Chunked transfer encoding.

INTRODUCTION

The worldwide web is an amazing place. You can find the answer to almost any question in less than ten mouse clicks. Many organizations provide critical operational data through websites on an intranet. Mining this data could provide many interesting conclusions for a SAS analyst. The data on these web pages would be a welcome addition to data warehouses. We need to find a way to automatically extract important data from web pages so that we can load it into a data warehouse.

SAS provides a method for extracting data from web pages – the URL access method of the FILENAME statement. This method makes it easy to read data from web pages without having to understand how web browsers communicate with web servers. Unfortunately, the URL access method only works with simple web pages. For example, web sites that have login screens use techniques that are not supported by the URL access method.

This paper describes the process the author uses to extract data from a Hitachi Global Storage Technologies intranet website that contains important yield information about the disk drive assembly and test process. The providers of this data decided to share it only through a web interface. The author had been using the URL access method, but this SAS program stopped working after the web developers released a new web server application. The goal of the SAS program is to automatically extract all of the yield and detractor information from the website on a daily basis and store it in a data warehouse. This data warehouse was described in previous SUGI papers (References 1 and 2).

The new web application has the following flow. From the welcome page, clicking on the 'Enter' link takes you to a login screen. You enter your user ID and password and click 'Log On', then you can see the main navigation menu. From here, you select a product group from the left pane and navigate through menus, specifying Disk drive factory, date, and disk drive model. Finally, you can view a yield report like the one shown in Figure 1.

The solution described here uses the SOCKET access method of the FILENAME statement to communicate with a web server. Another solution using the CURL open-source software is also described. Before these are shown, you need to have a basic understanding of how web browsers work.

HOW THE WEB WORKS – URLS, HTML, AND HTTP

A good way to understand how web browsers work is to think about 3 basic questions:

1. How do you identify a particular object on the web to retrieve, or what is the unique identifier of a web page?
2. How do you request the content from a server, and how will the server send it to you?
3. In what format will the content be sent so that it will be interpreted the same by all browsers?

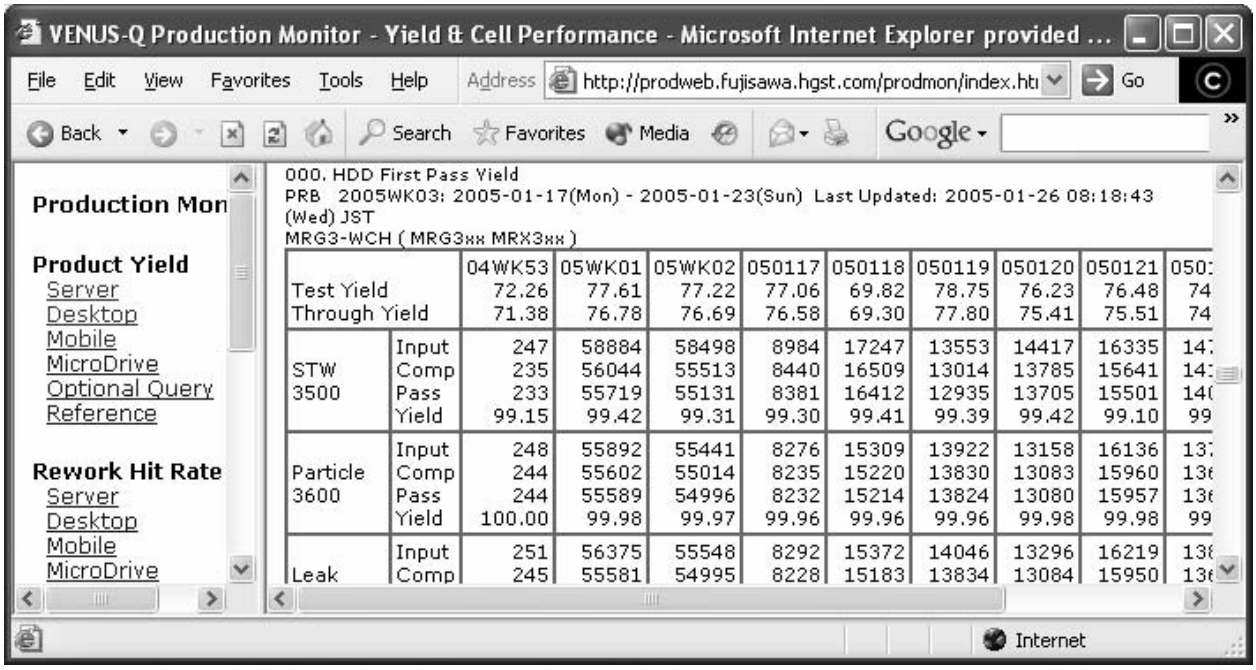
The answer to the first question is Universal Resource Locator (URL). The answer to the second question is Hypertext Transfer Protocol (HTTP). The answer to the third question is Hypertext Markup Language (HTML).

UNIVERSAL RESOURCE LOCATOR - URL

A URL is the name that is displayed in or you type into the address bar in your browser. For example, the URL for the SAS Corporate page at http://www.sas.com/corporate/index.html has the following components:

- http – Protocol, other examples are ftp and mailto
- www.sas.com – network location in hostname or IP address form
- /corporate/index.html – the path to the resource on the server, in this case a file named index.html

This page is an example of a static web page. This means that there is actually a file on the server in this directory called index.html, and the server sends the same file to everyone who requests this URL.



		04WK53	05WK01	05WK02	050117	050118	050119	050120	050121	050
Test Yield		72.26	77.61	77.22	77.06	69.82	78.75	76.23	76.48	74
Through Yield		71.38	76.78	76.69	76.58	69.30	77.80	75.41	75.51	74
STW 3500	Input	247	58884	58498	8984	17247	13553	14417	16335	14
	Comp	235	56044	55513	8440	16509	13014	13785	15641	14
	Pass	233	55719	55131	8381	16412	12935	13705	15501	14
	Yield	99.15	99.42	99.31	99.30	99.41	99.39	99.42	99.10	99
Particle 3600	Input	248	55892	55441	8276	15309	13922	13158	16136	13
	Comp	244	55602	55014	8235	15220	13830	13083	15960	13
	Pass	244	55589	54996	8232	15214	13824	13080	15957	13
	Yield	100.00	99.98	99.97	99.96	99.96	99.96	99.98	99.98	99
Leak	Input	251	56375	55548	8292	15372	14046	13296	16219	13
	Comp	245	55581	54995	8228	15183	13834	13084	15950	13

Figure 1 - Sample Yield Report

Many web pages are dynamically generated. Instead of being an actual file in a directory on the server, the server runs a program that generates the HTML content at that moment and sends it back to your web browser. For example, go to Yahoo, click Finance, enter 'IBM' in the Symbol box and click 'Go', you will go to a page that has this URL displayed in the address bar: http://finance.yahoo.com/q?s=IBM. This URL has the following components:

- http – Protocol, other examples are ftp and mailto
- finance.yahoo.com – network location in hostname form
- /q – path to the resource, in this case a program named 'q'
- s=IBM – query string, a set of variable name/value pairs separated by the '?' symbol that is passed to the program. In this case, only one variable named 's' with value 'IBM' is passed to program 'q'.

HYPERTEXT MARKUP LANGUAGE - HTML

Hypertext Markup Language (HTML) defines how the content in the web page should be formatted by your web browser. Because HTML is a markup language, it has many tags that control formatting, like <html>, <table>, <p>, and so on. Most browsers let you view the actual HTML for a web page. In Internet Explorer, you can view HTML by selecting View -> Source from the command bar. HTML is well documented. You can start at the official site, http://www.w3.org/MarkUp, or get one of the many books about HTML.

For the present task of automating web requests, you don't care how the content will be displayed in a browser; you only care about extracting certain pieces of data from the HTML page returned from the server. You do this with a SAS Data step that uses functions like INDEX and SCAN to locate HTML tags like , <table>, <tr>, and <td>, then parses out the data elements with functions like SCAN and SUBSTR. This usually requires intermediate to advanced Data step programming skills. Explaining how to write such a Data step is beyond the scope of this paper. SAS Institute offers several Data Step programming classes. The Online Documentation also has good information on this subject.

HYPERTEXT TRANSPORT PROTOCOL - HTTP

Hypertext Transport Protocol defines how a browser sends a request for a web page to a web server, and also how the server sends the HTML back to the browser. HTTP is also well defined and documented. You can find the specification for HTTP version 1.1 (the latest version) on the Internet at ftp://ftp.rfc-editor.org/in-notes/rfc2616.pdf (Reference 3). You should also get a good book about HTTP, such as *HTTP Developer's Handbook* (Reference 4). Luckily, you don't need to understand everything about HTTP to automate web page extraction with the techniques described in this paper. The quick overview in this section should be enough to get started.

An HTTP transaction consists of one request sent to a server, followed by one response from the server. Both the response and request must be in a precise format. An HTTP request consists of a request line, followed by HTTP header lines, followed optionally by content. An HTTP response consists of a status line, followed by HTTP header lines, followed optionally by content lines.

Here is an example of an HTTP request to get the default document at support.sas.com:

```
GET / HTTP/1.1
```

```
Host: support.sas.com
Connection: Close
```

Note that this HTTP request consists of a request line, two HTTP header lines, and no content lines. This request line, like all HTTP request lines, contains 3 items separated by blanks. The first item is the request method, in this case GET. The second item is the location of the resource on the server. The special character / tells the server to return the root document. In this case the web server will return the default, or home page for the site. The third item is the version of HTTP being used, in this case HTTP/1.1.

Here is the response from the web server to this request:

```
HTTP/1.1 200 OK
Date: Fri, 28 Jan 2005 23:27:31 GMT
Server: Apache/1.3.29 (Unix) mod_jk/1.2.0
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html

1000
<html>
<head>
<title>SAS Customer Support Center</title>
{many more lines of HTML}
```

Note that this HTTP response consists of a status line, five HTTP header lines, and many content lines. This status line, like all HTTP status lines, contains 3 items separated by blanks. The first item is the version of HTTP being used, in this case HTTP/1.1. The second item is the response status code, a 3-digit number that tells you the outcome of the server's attempt to fulfill your request. The third item is a short description of the status code. The following sections describe some important HTTP topics in more detail.

REQUEST METHODS

HTTP 1.1 defines eight request methods, but you only need to know about two of them – GET and POST. The most popular request method used by web clients is GET. This is the type of request your browser uses when you click on a link or type a URL into the browser's address bar.

GET is also the default method used by HTML forms. Whenever you enter text into an entry field, select a radio button or checkbox, or click a button like 'Submit' on a web page, you are interacting with an HTML form. HTML forms are started by the `<form>` tag. This tag has an attribute called `method` that can take values of `get` and `post`. When Get is used, each form field name and value is built into a query string as described earlier in the discussion of Universal Resource Locators, and the browser issues a GET request to retrieve the content specified by the `action` attribute of the `<form>` tag. All form variable names and values are included in the resource field on the HTTP request line. GET does not send any data in the content section after the HTTP headers.

For example, on the Google home page, when you type SAS DATA WAREHOUSE in the text box and click Google Search, the browser sends a GET request. Note that the query string is sent as part of the resource name:

```
GET /search?hl=en&q=sas+data+warehouse HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Referer: http://www.google.com/
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: www.google.com
Connection: Keep-Alive
```

The POST method is also used to request content from a web server. Most browsers use the POST method only when the `<form>` tag specifies `method=post`. In this case, the query string is sent in the content section after the HTTP headers. If the Google search had specified the POST method instead of GET, the request would look like:

```
POST /search HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
Referer: http://www.google.com/
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: www.google.com
Connection: Keep-Alive
Content-Length: 26
Content-Type: application/x-www-form-urlencoded

hl=en&q=sas+data+warehouse
```

Note that the content section that contains the query string is separated from the HTTP headers by a null line. Also note that two additional HTTP headers, `Content-Length` and `Content-Type`, are required. This lets the web server know how much content data to expect, and how to interpret it.

RESPONSE STATUS CODES

The HTTP standard defines several 3-digit response status codes, grouped into five categories: Information (100-199), Success (200-299), Redirection (300-399), Client error (400-499), and Server error (500-599). The status code you want to see is 200 (OK). This means the request was acceptable and the server processed it successfully. If there is a syntax error in your HTTP request, expect to see status code 400 (Bad Request). You've probably seen status code 404 (Not Found) displayed in your browser when a requested document does not exist, and status code 401 (Unauthorized) when you cancel out of a password dialog box.

One status code you should know about is 302 (Object Moved). This is one of the Redirection status codes, and it is used when the Web server wants to inform the client that the requested resource is located somewhere else. You would think that redirection would mostly be used to point stale links to the correct resource, but it is actually most often used when redirection is desired as a normal part of the Web application, such as testing to see if a browser will accept cookies. When a Web server sends a 302 status code, it must also send the new URL for the resource in a `Location` header. The browser then issues a second request to retrieve the resource using this new URL. This happens automatically, so that you do not notice anything different by looking at the browser window.

HTTP HEADERS

The HTTP 1.1 standard defines dozens of headers. Fortunately, you only need to know about a few of them to script web requests. The headers fall into 4 categories: headers only used in requests; headers only used in responses; and general and entity headers, which can be used in either requests or responses.

Request headers are used only in an HTTP request. The only required header in an HTTP 1.1 request is the `Host` header, which identifies the hostname you are trying to reach. The `Accept, Accept-Charset, Accept-Encoding` and `Accept-Language` headers tell the server what content types, character encodings, content encodings, and content languages the browser is willing to accept. The `User-Agent` header identifies the type of Web browser making the request. Some server applications send back different HTML for different user agents.

Response headers are used only in an HTTP response. The `Server` header identifies the Web server software, like Apache or Microsoft IIS. The `Location` header was mentioned earlier in the discussion about redirection.

General headers can be used in requests or responses. Many of these are related to caching of resources. The `Connection` header indicates whether the connection between the client and server should be or has been closed or kept open. The `Date` header indicates the system time on the client or server.

Entity headers give information about the content of an HTTP message. Since a GET request does not send content, no entity headers are used. The `Content-Type, Content-Length, Content-Encoding` and `Content-Language` headers indicate the type, length in bytes, type of encoding, and language of the content.

COOKIES

Cookies are essential to the Web for state management. HTTP is often called a *stateless* protocol. This means that each transaction is atomic, and there is nothing required by HTTP that associates one request with another. This makes HTTP simple to use, but limits its usefulness. For example, a website like Amazon that has a shopping cart feature needs a way to keep track of each user and the history of their session at the site.

State management in HTTP is enabled by the use of Cookies. A cookie is a token with a name and value that is sent in HTTP requests and responses. A web server sends a cookie to the client with the `Set-Cookie` header, and a client sends a cookie to a web server with the `Cookie` header. In the Expedia example earlier, you saw that the web server sent a cookie with this header:

```
Set-Cookie: MC1=GUID=2204A39CCBE1442CB4CD20B9875AE6C4; expires=Fri, 01-Jan-2010
08:00:00 GMT; domain=.expedia.com; path=/
```

In this example, the cookie name is `MC1` and its value is `GUID=2204A39CCBE1442CB4CD20B9875AE6C4`. The other tokens named `expires`, `domain`, and `path` are attributes of the cookie. A web client must send all cookies accepted from a web server back to the web server in subsequent requests with the `Cookie` header. Multiple cookie name/value pairs are separated by semicolons. Since three cookies were set in the Expedia example, the next request to the site would include this header:

```
Cookie: COOKIECHECK=1; ipsnf=us|1; MC1=GUID=2204A39CCBE1442CB4CD20B9875AE6C4
```

CHUNKED TRANSFER ENCODING

It is quite common for content to be returned from a web server in a format called Chunked Transfer Encoding. Instead of sending the `Content-Length` header and specifying the entire length of content, the server will send the `Transfer-Encoding: chunked` header. In this case, the content is sent in chunks, and the length of each chunk in hexadecimal bytes is sent before the chunk. Here is an example of this from support.sas.com:

```
HTTP/1.1 200 OK
Date: Wed, 02 Feb 2005 23:59:50 GMT
Server: Apache/1.3.29 (Unix) mod_jk/1.2.0
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html

1000    {chunk size 1000x = 4096 bytes}
```

```
<html>  {start of first chunk}
<head>
{more lines of HTML in this chunk}
         <td><br> <FORM method="get" action="http://search.sas.com/suppquery.html"
style="padding:0;margin      {end of first chunk}
1000    {chunk size 1000x = 4096 bytes}
:0">    {start of second chunk}
               <span class="search">Search Customer Support</span>
```

This pattern of alternating chunk lengths and chunk data continues until the content is all sent, then the web server sends a final chunk length of zero and an empty chunk. This signals the Web client that the content is finished. Chunked transfer encoding is useful for sending dynamically generated content where the web server does not know the length of the entire content at the time when the headers are created. It also allows the browser to render and display the page as each chunk is received, which makes the initial display much quicker for large pages. However, if you are scripting web requests and reading raw HTTP responses, you need to parse the content carefully. You see from this example that the chunk boundaries usually fall in the middle of an HTML line. Your HTML parsing algorithm probably assumes that the original HTML lines are intact, so you need to process the chunks and put split HTML lines back together before you parse the HTML.

READING WEB PAGES WITH FILENAME URL

SAS provides a special form of the FILENAME statement for accessing web sites, known as the URL access method. This access method generates the HTTP requests and responses, so you only need to worry about parsing the desired information from the returned HTML. For example, to retrieve the HTML for the support.sas.com web page into a SAS data set, you could run the following program. In a real program, you would use functions like INDEX and SCAN to locate the parts of the HTML of interest, then parse out the information you need. The debug option tells SAS to display the HTTP headers in the SAS log. Note that many web pages have lines longer than the default 256 characters, so you can use the lrecl option to get the entire line without truncation.

```
filename junk url 'http://support.sas.com' debug lrecl=8192;
data http;
 infile junk length=len;
 input record $varying8192. len;
 run;
```

FILENAME URL works well for many websites. However, it has some key limitations. First, only the GET request method is supported. If you need to script access to a resource that uses the POST method, FILENAME URL will not work. Since most applications protect sensitive data by prompting for user ID and password, and use the POST method to send the user ID and password to the server, FILENAME URL cannot be used for these.

A second limitation of FILENAME URL is that it purposely does not let you read or write HTTP headers, to make it simple to use. Since many web applications require cookies for state management, FILENAME URL will not work.

HTTP VIEWER PROGRAMS

Now that you know a little bit about HTTP, and are eager to automate data retrieval from your favorite web sites, how do you find out what HTTP is sent and received for the specific sequence of web pages you want to access, so you can replicate it in a SAS program? There are several shareware and commercially available programs that let you observe the HTTP statements that the browser sends to a web server, and see the HTTP statements that are returned from the web server. Three such viewers are httpExplorer (http://www.overture-computing.com), IEWatch (http://www.iewatch.com), and HttpWatch (http://www.httpwatch.com).

Here is a description of httpExplorer. When you open the program, the top section contains an Internet Explorer browser, and the bottom section contains a list of HTTP requests for all resources loaded with a web page. If you double-click one of the resources in the lower section, the detailed HTTP request and response messages are loaded into the upper section. If you click on the Request Data or Response Data tabs, you can see the raw binary stream for the request and response.

One handy tip for using httpExplorer is that you should delete cookies and locally cached web pages from your hard drive before you start the program. Otherwise, saved cookies will be sent in HTTP requests before the web server sets them in an HTTP response, which is very confusing. And under certain circumstances, the web server will send a 304 status code and no HTML content if the page was visited recently and cached locally. You can erase cookies by starting Internet Explorer and selecting Tools, Internet Options, then Delete Cookies. You can delete locally cached web pages by selecting Tools, Internet Options, then Delete Files from Internet Explorer.

httpExplorer has a few limitations. First, it does not handle content encoding, so if the web server returns compressed content, you won't be able to see the HTML. httpExplorer also does not handle HTTPS, or HTTP with Secure Socket Layer (SSL) security. The author has not tried other viewer programs, and does not know if they have these same limitations.

READING WEB PAGES WITH FILENAME SOCKET

Sockets are a feature of TCP/IP that let computers communicate with each other. A web client can communicate with a web server by opening a socket connection with the server. Web servers listen for HTTP messages over TCP/IP port 80, which is the standard port for HTTP. SAS provides an Access Method of the FILENAME statement called SOCKET to let SAS programs send and receive data over TCP/IP sockets from a Data step

Here is a simple example showing a Filename statement with the Socket Access Method, and a Data step that sends an HTTP request to a web server and receives an HTTP response from the server. In this example, the HTTP response is displayed in the SAS log.

```
filename http socket "support.sas.com:80" termstr=LF lrecl=8192;
data _null_;
 infile http length=len;
 file http;
 if _n_=1 then put
   'GET / HTTP/1.1' /
   'Host: support.sas.com' /
   'Connection: close' / ;
 input;
 file log;
 put _infile_;
 run;
```

There are several features to note about this FILENAME statement. First, you must specify the port number after the host name. Also, you tell SAS how to parse the response from the server into records in the Data step with the `termstr` option. Typically you will use either LF, which splits records whenever it sees hexadecimal character '0A'x, or CRLF, which splits records at '0D0A'x. It seems to the author that most web servers send HTTP headers delimited by '0D0A'x, while records in the content are delimited by '0A'x, but this may vary depending on the operating system of the server and possibly the client. Finally, the `termstr` option can be used to specify the record length, since many HTML lines are longer than the default 256 characters.

This Data step is different than what you usually see. Most Data steps in SAS programs read data from a file or write data to a file, but not both to the same file in the same Data step. This data step writes to and reads to one socket connection. This is why the fileref HTTP is used on both an INFILE and FILE statement. On the first iteration of the Data step, the entire HTTP request is written to the socket, and then an HTTP response record is read from the socket at each iteration of the Data step.

You should always include a `Connection: close` header in a FILENAME SOCKET request to tell the server to close the connection after the response is sent. The default behavior of HTTP 1.1 is persistent connections, and SAS will wait for up to several minutes until it times out if you don't include this header. You should not include a `Accept-Encoding` header unless you are prepared to handle compressed content. If you omit this header, web servers will return the content in clear text.

MACRO %HTTP – A HANDY TOOL FOR READING WEB PAGES

Here is a macro that simplifies reading web pages with FILENAME SOCKET. The macro submits an HTTP request to a web server and writes the request and response to a file on your computer. The macro determines if the response is sent with chunked transfer encoding, and reassembles the HTML content into its original format if it was.

```
%macro http(http=%nrstr(), save_file=, hostname=, delimiter=%str(|),
   fileref=browser, termstr=CRLF);

%let str1=; %let str2=;
%do %while (%index(&http,&delimiter)>0);
 %let pos=%index(&http,&delimiter);
 %let str1=&str1%qsubstr(&http, 1, &pos-1)/;
 %let str2=&str2%qsubstr(&http, 1, &pos-1)'0d0a'x;
 %let http=%qsubstr(&http, &pos+1, %length(&http)-&pos);
 %end;

filename &fileref socket "&hostname:80" termstr=&termstr lrecl=8192;
data _null_;
 retain chunked;
 length rec2 $10000 chunked $1;
 infile &fileref length=len;

 if _n_=1 then do;
  file &fileref;
  put %unquote(&str1);
  file &save_file;
  put '>>> HTTP Request:' '0d0a'x;
  put %unquote(&str2);
  put '0d0a'x '<<< HTTP Response:' '0d0a'x;
  end;
```

```
    file &save_file recfm=n;
    input record $varying8192. len;
    put record $varying8192. len '0d0a'x;
    if record='Transfer-Encoding: chunked' then chunked='Y';

    if len=0 and chunked='Y' then do while (1); %*** look for & process chunks;
     input chunksize $varying8. len /  record $varying8192. len;
     if len ne 0 then do;
      l2=len+countc(record, '0a'x);
      rec2=tranwrd(record, '0a'x, '0d0a'x);
      put rec2 $varying10000. l2;
      end;
     else stop;  *** found null record at end of file, end loop;
     end;
    run;
   %mend http;
```

Here is a sample call of this macro:

```
   %http(
     hostname=support.sas.com,
     save_file='c:\support_sas.txt',
     http=%nrstr(
      'GET / HTTP/1.1' |
      'Host: support.sas.com' |
      'Connection: Close' | )
        );
```

Here is the file c:\support_sas.txt created by the macro:

```
   >>> HTTP Request:
   GET / HTTP/1.1
   Host: support.sas.com
   Connection: Close

   <<< HTTP Response:
   HTTP/1.1 200 OK
   Date: Thu, 03 Feb 2005 17:22:25 GMT
   Server: Apache/1.3.29 (Unix) mod_jk/1.2.0
   Connection: close
   Transfer-Encoding: chunked
   Content-Type: text/html

   <html>
   <head>
   <title>SAS Customer Support Center</title>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
   {many more lines of HTML}
```

Here are some usage notes for this macro. The three required parameters are Hostname, Save_file, and HTTP. Argument Hostname specifies the hostname or IP address of the server to connect to; don't put this name in quotes. Argument Save_file is the name of a local file to save the HTTP to, in single or double quotes. Argument HTTP contains the HTTP request. This argument must be enclosed in the %nrstr macro function. All lines, including the last, must be enclosed in single or double quotes and end with a separator character that must not appear anywhere else in the HTTP request. The default separator character is '|', which is on the key above the Enter key. If you need to use this character in the request, specify a different separator character with the Delimiter argument. The macro uses CRLF as the termination string option on the FILENAME statement. This should work well for most web pages. If you find a page that does not work with the default settings, for example, it returns HTTP response headers but no HTML content when some was expected, try adding `termstr=LF` to the macro call.

Because the HTTP request is sent with a PUT statement in a Data step, you have great flexibility in assembling the strings that make up any of the request lines. This greatly simplifies submitting requests containing macro variables, which is required to script web page retrieval. This is demonstrated in an example in the next section.

The macro does not attempt to handle content encoding (like Gzip) or HTTPS, (HTTP with Secure Socket Layer). If you need to use either of these, see the discussion about Curl in a later section.

NAVIGATING THE YIELD WEBSITE WITH THE %HTTP MACRO

Now that you understand how HTTP works, have an HTTP viewer program that shows you HTTP requests and responses for each step of the Web application, and have a SAS tool to send and receive HTTP messages, you are ready to write a SAS program to automate extracting data from Web sites. Here is some information about how the author did this for the Hitachi Web application described in the introduction.

When you click on the Enter link on the welcome page, httpExplorer shows that the request is for a resource called

/prodmon/index.html, and the response sets a cookie named JSESSIONID and redirects you to resource /prodmon/login.jsp, which displays the login page. We need to save the cookie in a macro variable so we can send it in the next request. Here is the %HTTP call to the first resource:

```
%http(
  hostname=prodweb.fujisawa.hgst.com,
  save_file='c:\mysas\reports\prodweb1.txt',
  http=%nrstr(
   'GET /prodmon/index.html HTTP/1.1' |
   'Host: prodweb.fujisawa.hgst.com' |
   'Connection: Close' | )
    );
```

Here is the file it created:

```
>>> HTTP Request:
GET /prodmon/index.html HTTP/1.1
Host: prodweb.fujisawa.hgst.com
Connection: Close

<<< HTTP Response:
HTTP/1.1 302
Date: Thu, 03 Feb 2005 21:48:38 GMT
Server: Apache/2.0.52 (Unix) mod_jk2/2.0.3-dev
Pragma: No-cache
Cache-Control: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Set-Cookie: JSESSIONID=F866B80E5D5B087A2A1BE7A814C9FD7B; Path=/prodmon
Location:
http://prodweb.fujisawa.hgst.com/prodmon/login.jsp;jsessionid=F866B80E5D5B087A2A1BE7
A814C9FD7B
Content-Type: text/plain; charset=ISO-8859-1
Content-Length: 0
Connection: close
```

Here is a Data step to read this file and store the cookie in macro variable &session_ID:

```
data _null_;
 infile 'c:\mysas\reports\prodweb1.txt' lrecl=8192 length=len;
 input record $varying8192. len;
 if scan(record,1,' ')='Set-Cookie:' then
   call symput('session_ID', scan(record,2,' '));
 run;
```

httpExplorer showed that the redirected call to resource /prodmon/login.jsp did not return any cookies, so we don't need to submit a request to this resource from our SAS program. The next step is to submit our user ID and password. httpExplorer shows that this request uses the POST method for resource /prodmon/j_security_check. Here is the macro call to do this from SAS. Note that the cookie in the macro variable is sent in a `Cookie` header, and also note the extra separator after the `Content-Length` header that creates the required null line.

```
%http(
  hostname=prodweb.fujisawa.hgst.com,
  save_file='c:\mysas\reports\prodweb2.txt',
  http=%nrstr(
   "POST /prodmon/j_security_check HTTP/1.1" |
   'Host: prodweb.fujisawa.hgst.com' |
   'Connection: Close' |
   "Cookie: &session_ID" |
   "Content-Type: application/x-www-form-urlencoded" |
   "Content-Length: 47" |
   |
   'j_username=helf&j_password=notmypw&Logon=Log+On' | )
    );
```

Here is the file created for this HTTP transaction:

```
>>> HTTP Request:
POST /prodmon/j_security_check HTTP/1.1
Host: prodweb.fujisawa.hgst.com
Connection: Close
Cookie: JSESSIONID=F866B80E5D5B087A2A1BE7A814C9FD7B;
Content-Type: application/x-www-form-urlencoded
Content-Length: 47

j_username=helf&j_password=notmypw&Logon=Log+On

<<< HTTP Response:
HTTP/1.1 302
```

```
Date: Thu, 03 Feb 2005 21:59:30 GMT
Server: Apache/2.0.52 (Unix) mod_jk2/2.0.3-dev
Location: http://prodweb.fujisawa.hgst.com/prodmon/index.html
Content-Type: text/plain; charset=ISO-8859-1
Content-Length: 0
Connection: close
```

You can see that the server redirects the web client to resource /prodmon/index.html, which returns the main navigation menu. httpExplorer shows that this resource sets a second cookie, so the program uses %http to request this resource, then runs a Data step like the previous one to save the second cookie in another macro variable.

At this point, we have successfully completed the user authentication process, and are ready to request and receive report pages. httpExplorer shows that all of the HTTP requests from the selection menu use the same server program called /prodmon/yield/servlet/yieldReport, but with different values for parameters in the query string. The SAS program calls the %HTTP macro several times to find out all combinations of product, family, manufacturing site, and production week, and stores these values in macro variables. The program issues this call to retrieve the HTML for the yield report displayed in Figure 1:

```
%http(
  hostname=prodweb.fujisawa.hgst.com,
  save_file='c:\mysas\reports\prodweb4.txt',
  http=%nrstr(
   'GET /prodmon/yield/servlet/yieldReport?type=process&area=HDD&variation=new&'
     "family=&family" '&' "product=&product" '&' "site=&site"
     '&action=showReport&' "week=&week" '&viewLevel=Summary HTTP/1.1' |
   'Host: prodweb.fujisawa.hgst.com' |
   "Cookie: &session_ID &session_ID_sso" |
   'Connection: Close' | )
     );
```

Here is the file created by this macro call. Note that the resource name was broken into several strings. The four macro variables were specified inside double quotes, since macro variables are not resolved inside single quotes. The rest of the strings without macro variables were specified inside single quotes. This prevents the ampersand symbol used to separate name/value pairs from being interpreted as part of a macro variable name.

```
>>> HTTP Request:
GET
/prodmon/yield/servlet/yieldReport?type=process&area=HDD&variation=new&family=Mobile
&product=Moraga-A&site=PRB&action=showReport&week=2005WK03&viewLevel=Summary
HTTP/1.1
Host: prodweb.fujisawa.hgst.com
Cookie: JSESSIONID=F866B80E5D5B087A2A1BE7A814C9FD7B;
JSESSIONIDSSO=854135B008AD8CB92D81182F8D791BC6;
Connection: Close

<<< HTTP Response:
HTTP/1.1 200
Date: Thu, 03 Feb 2005 23:02:26 GMT
Server: Apache/2.0.52 (Unix) mod_jk2/2.0.3-dev
Pragma: No-cache
Cache-Control: no-cache
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Type: text/html;charset=ISO-8859-1
Connection: close
Transfer-Encoding: chunked

<html>

<head>
  <title>Product Yield</title>
  <link rel='stylesheet' type='text/css' href='/css/prodweb.css'></link>
  <link rel='stylesheet' type='text/css' href='/prodmon/yield/thintable.css'></link>
{many more lines of HTML}
```

The SAS program extracts the desired data elements from the HTML of the yield and detractor reports, then saves them in SAS data sets in the SAS data warehouse.

READING WEB PAGES WITH CURL THROUGH FILENAME PIPE

After developing this solution to automate retrieval and extraction of data from web pages, the author found out about the CURL program from David Ward's SUGI 28 paper (Reference 6). CURL is an open source (meaning free) command line program that lets you retrieve data from Web servers. It has many powerful features, including support for HTTPS, although the author has not tested this yet.

CURL is available at http://curl.haxx.se, or from several mirrors located in the US. To use CURL from SAS, download one of the binary packages, with or without SSL support. If you download one with SSL support, also download the OpenSSL library/DLL package from the same page.

CURL has dozens of command line options that give you complete control over how the HTTP request is constructed. The HTTP response is displayed in the command line window, or you can specify an option to write the response to a file. Like all command line programs, you can call CURL from a SAS program as an unnamed pipe with the FILENAME PIPE statement, and process the CURL output in a Data step. Here is an example of using CURL from SAS to retrieve the support.sas.com main page and display the HTTP result in the SAS log:

```
filename curl pipe "c:\progra~1\curl\curl -i http://support.sas.com" lrecl=8192;
data _null_;
 infile curl length=len;
 input;
 put _infile_;
 run;
```

INTERNET MACROS BY IOPUS

It may be possible to automate complex web page retrieval without learning HTTP at all. Take a look at a software product called Internet Macros by iOpus (http://www.iopus.com/iim/). This software automates web page retrieval by first going through a keystroke recording process for the web pages you access, then playing the macro from a command line or batch job to retrieve the pages and save them locally. According to the Features list on the website, the top-of-the-line $499 Scripting Edition has advanced features like variables, command line support, scripting interface, web page data extraction, and the ability to return web page data to a script for further processing. It may be possible to use the software to automate the retrieval of web pages, then use SAS to read in files saved by Internet Macros and process the data in SAS. The author has not tried this software yet.

CONCLUSION

Being able to automate data extraction from web pages beyond the reach of FILENAME URL opens up a large new data source for Data Warehouse designers. This paper showed how HTTP works, described an HTTP viewer program that displays HTTP requests and responses for each step of the Web application, and explained how to process HTTP transactions in SAS with FILENAME SOCKET and the CURL open source program.

REFERENCES

1. Rutledge, Robert A., Tai, Linda H., and Helf, Garth W., “Just Enough Database for Manufacturing Yield Analysis”, *Proceedings of the 24th SAS Users Group International Conference*, 1999, http://www2.sas.com/proceedings/sugi24/Dataware/p116-24.pdf
2. Rutledge, Dr. Robert A., “Data Warehousing for Manufacturing Yield Improvement”, *Proceedings of the 25th SAS Users Group International Conference*, 2000, http://www2.sas.com/proceedings/sugi25/25/dw/25p134.pdf
3. Fielding, R., et al, “Hypertext Transport Protocol – HTTP/1.1”, *The Internet Engineering Task Force, Request For Comments 2616*, 1997, ftp://ftp.rfc-editor.org/in-notes/rfc2616.pdf
4. Shiflett, Chris, “HTTP Developer’s Handbook”, *Sams*, 2003
5. Ward, David L., “You Can Do THAT with SAS Software? Using the socket access method to unite SAS with the Internet”, *Proceedings of the 2000 Northeast SAS Users Conference*, 2000, http://www.nesug.org/html/Proceedings/nesug00/iw/iw5006.pdf
6. Ward, David , “SAS and the Internet for Programmers”, *Proceedings of the 28th SAS Users Group International Conference*, 2003, http://www2.sas.com/proceedings/sugi28/128-28.pdf

ACKNOWLEDGMENTS

I would never have been able to figure this out on my own. I was able to use the SOCKET access method to simulate a web browser only after reading two excellent papers by David Ward. See References 5 and 6.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Garth Helf
Hitachi Global Storage Technologies
5600 Cottle Road
San Jose, CA 95193
Phone: 408-717-7514
Email: Garth.Helf@HitachiGST.com