Recording Links Followed In A Single HTML Document

James Blustein

Technical Report CS-2002-06

September 19, 2002

Faculty of Computer Science
6050 University Ave., Halifax, Nova Scotia, B3H 1W5, Canada

Recording Links Followed In A Single HTML Document

James Blustein*
Faculty of Computer Science, Dalhousie University
Halifax, Nova Scotia
B3H 3R3, Canada
E-mail: jamie@cs.dal.ca

19 September 2002

*In press at *The Perl Journal* in 1999. Rejected in 2002.

Abstract

This report describes in detail a combination of tools (written in Perl, Javascript and the hypertext markup language (HTML) that were used to record the links users followed within a single web page.

The method was developed specifically for use with the Perl 5 language and the Netscape World Wide Browser (version 4 for Unix) but can be easily adapted to other browsers. The main drawback to the method is that users must click the 'back' button twice to return to the previous link. It should be a simple matter to eliminate that drawback using today's browsers.

Contents

List of Figures

Chapter 1

Introduction

Sometimes web log files are not enough to track where a user has really gone in your site. As part of a hypertext experiment [1] I needed to know exactly which links users followed. With a log file or mini server I would only be able to tell which files were loaded but not which links users followed. This information is especially critical when you need to answer questions such as 'do users follow "table of contents" links or just scroll down to the content?' Of course even if you know which links someone followed you may not know why they chose to follow it or if it was helpful to them, but that is another story.

1.1 Review of WWW Links

So how can we know which links a user follows when reading a single file? A brief review of how links on the World Wide Web (WWW) work is in order. The WWW uses a client-server model, meaning that when a user clicks on a link, the browser software (Netscape for instance) asks the server (a computer somewhere on the network) for a file in Hypertext Markup Language (HTML) format. The browser receives the file file, interprets the HTML markup to present to the user. Some of the markup is is about links and some about presentation.

Hypertext links in the HTML are from anchors (the '`a`' element) and go to files or other anchors within HTML files. When the destination of the link is a file then the browser requests that file from the server* When the

*Savvy readers will notice that I'm not mentioning caches (or non-GUI browsers). That is okay because this is only the basic overview of the WWW necessary to understand the code.

destination is an anchor in an HTML file that file is retrieved, rendered, and displayed in an appropriate fashion.

1.2 Using CGI — slow but portable

So how can we record which links a user follows? Obviously we need to have some code executed every time a link is followed. The code will record which link was followed, by writing to a file. My code is executed on the server.

The most straightforward way to track links (and one that will work for all WWW browsers) is to run a CGI* program in place of a link. Part (a) of Figure 1.1 shows an ordinary link, Part (b) shows an equivalent link accomplished with a call to to a CGI program named `tracker`. The program can write a record listing th `href` and `name` arguments that were passed to it. It could even record the time that the link was selected. The program would also have to make the browser act as though the original link (the `href` from Figure 1.1 part a) had been followed. A CGI program program can do that by issuing a redirect instruction which will cause the browser to load the document from the network and process as usual. That method is portable and simple.

However that simple method can be very slow. The redirect requires the server to reload the entire document, even if it is already in the browser's cache. The delay might not be noticeable for small documents but it is unacceptably slow for many documents. Hypertext and human factors expert Jakob Nielsen [3] says that following links should take no longer than one second.

*CGI stands for Common Gateway Interface. See ⟨`URL:http://hoohoo.ncsa.uiuc.edu/cgi/interface.html`⟩ for details.

```
<a  name="here"  href="#somewhere"  >click here</a>
```

(a) Ordinary link

```
<a  name="here"
    href="/cgi-bin/tracker?name='here';href='#somewhere'"
    >click here</a>
```

(b) CGI link

Figure 1.1: Links in HTML

Chapter 2

A Faster Way

The method I developed to record which links had been followed is fast but non-portable. I have used it with Netscape (version 4.04 for Solaris) and I expect that it can can be used (with some small modifications) with Internet Explorer. My method is in brief: use JavaScript (JS) to do two things at once, namely follow the link and execute the CGI program.

2.1 Use JavaScript — Fast where it works

I determined experimentally that the CGI program must produce output that the browser can render or it fails. So I create a hidden frame for the output to 'appear' in. A hidden frame has no size. (I specify either zero rows or zero columns, but not both. The browser I used did not work properly when the frame had both dimensions set at zero.)

Here is how it works: the HTML document (Figure 2.1) is for display to users. Figure 2.2 shows the frameset I use to make the hidden frame. I use a Perl program based on some code of Tom Christiansen's [2] to insert the JS calls (see Appendix A). JavaScript is a client-side programming language for changing the browsers behaviour. Using it is the only way I know, short of hacking the browser's own code, to make selecting a link do two things. What the JS code does is to run the CGI program

What the

```
onClick="go('%23ToC%3AS1.1', 'L000', 'description')"
```

means is that when the link is selected (by say clicking with the mouse) then execute the JS '`go`' function which is defined in the `head` and follow the `href`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>HTML file for users to see</title>
  <script type="text/javascript">
   <!-- // hide script from non-JS browsers
      function go(href, name, title) {
         top.hidden.location.href="/cgi-bin/tracker?n=" + name + "&" +
                                                    "h=" + href + "&" +
                                                    "t=" + title
         return true
      }
    // -->
  </script>
</head>

<body>
 <noscript>
   <strong>You <em>must</em> allow</strong>:
   <ul>
     <li><strong>Javascript</strong>, &amp;
     <li><strong>Cookies</strong>!
   </ul>
 </noscript>

 Here are some links:
 <ul>
  <li><a title="description" name="L000" href="#ToC:S1.1"
         onClick="go('%23ToC%3AS1.1', 'L000', 'description')"
        >to Contents section 1.1</a></li>

  <li><a title="description" name="L001" href="#ToC:S2.1"
         onClick="go('%23ToC%3AS2.1', 'L000', 'description')"
        >to Contents section 2.1</a></li>

  <li><a title="description" name="L002" href="#ToC:S3.1"
         onClick="go('%23ToC%3AS3.1', 'L002', 'description')"
        >to Contents section 3.1</a></li>
 </ul>
</body>
</html>
```

Figure 2.1: HTML document for users

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Frameset//EN">
<html>
<head>
 <title>Frameset for tracking links</title>
</head>

<frameset rows="100%,*">
 <frame src="doc.html"                          name="doc"     frameborder=0
        scrolling="yes">
 <frame src="/cgi-bin/tracker?t=''&f=''&n=''"  name="hidden"  frameborder=0
        scrolling="no">
 <noframes>
   <p>Sorry, you must allow frames to enable the link tracking.
      <a href="doc.html">This</a> is the non-tracking version.</p>
 </noframes>
</frameset>
</html>
```

Figure 2.2: The Frameset

as usual. As you can see from its definition, the function takes three parameters representing the `name`, `href`, and `title` from the anchor. To avoid possible complications: all of the non-alphanumeric characters have been replaced by their hexadecimal equivalents courtesy of the escape function in Lincoln Stein's `CGI.pm` CPAN module, and they are all quoted (I use single quotes here and double quotes around the function call). The semantics of what the function does are a bit tricky. It has the effect of executing the CGI program in the hidden frame. What it really does is to try to lead the document found at the location of the CGI program and display it in the hidden frame. It might not be necessary for '`go`' to return true but it seems prudent. By using JavaScript a HTML anchor can do two things at once.

2.2 CGI Cookies and other oddities

Section 2.3 shows the CGI program. When a link is followed we need to be able to determine which user followed it. I rely on CGI cookies to record which user it is.

A CGI cookie is a datum sent by the server to the client. The user might never know it is there. From time to time a CGI program may request the client to send a copy of the cookie back to it. Cookies can thus be used to identify users of websites.

2.3 CGI Program Listing

```
#!/usr/local/bin/perl -Tw

# Copyright (c) James Blustein 1996-2002
# 15 Sep 2002 (version 3.0a)
#   This version prints after recording (v2.1.2 redirected after recording)

# – pragmas
  use strict;
  use integer;
  use CGI qw(:standard); # for cookie code                                    10
  use CGI::Carp qw(carpout fatalsToBrowser);
  use diagnostics;

  use constant DEBUG_OUTPUT => 0;
  use constant VERBOSE => 0;
  use constant VERSION => 'link-track.pl v3.0a (15 September 2002)';
  use constant LOG_FNAME => 'log';
  use constant TEXTDIR => 'http://url.where.the/docs/seem/to/be/';

BEGIN {                                                                      20
    $| = 1;  # unbufferred output
    print "Content-type: text/plain\n\n";
    print "This is `" . VERSION . "'\n" if VERBOSE or DEBUG_OUTPUT;
}

{
   my($home);
   my($input) = new CGI;

   # – initialization                                                        30
   $home = "/actual/home/of/the/docs/";

   my(%cookie) = cookie('values');
   my($user)  = $cookie{'user'} || -255;
   my($dir)     = $home . "TPJ/Log";
```

```
my($logfile) = "$dir/u${user}/" . LOG_FNAME;

# – sanity checks
&error('No $home') if (!defined($home));

if ((1 > $user) || ($user > 5)) {
    warn " ** In testing mode **\n";
} else { # untaint
    $user =~ /^([1-5])/;
    $user = $1;
}

# – get paramaters from CGI call
my($doc) = $input->param('d');
my($to)  = $input->param('h');
my($from) = $input->param('n');
my($title) = $input->param('t');

# – open log file
# – open logfile by changing to directory (I don't know why this is necc.)
my($chdir_status) = chdir($dir);
&error("error from chdir($dir): $!\n") if (0 == $chdir_status);
$chdir_status = chdir("u${user}/");

&error("error from chdir(u${user}/) " .
       "(cwd=$dir): $!\n")
      if (0 == $chdir_status);

if (not((-f LOG_FNAME) and (-w LOG_FNAME))) {
    &error("Missing or incorrect logfile\nfilename = \"$logfile\"");
}
open (LOG, ">> " . LOG_FNAME)
    or &error("opening log file \"$logfile\": $!");


if (($from) and ($to)) {
    # – write entry to log file
    &log_entry(\*LOG, $from, $to, defined($title)?$title:"");
    close LOG or &error("closing \"$logfile\": $!");
```

```
        # – output to keep browser happy
        print "link-track OK\n";
        exit 0;
    } else {
        close LOG or &error("(no fields by the way) closing \"$logfile\": $!");
        &error("Can't find CGI fields ('from', 'to', etc.)\n");
    }
}


# – subroutines
sub html_quote {
    my($message) = @_;

    $message =~ s/&/&amp\;/g;
    $message =~ s/</&lt\;/g;
    $message =~ s/>/&gt\;/g;

    return $message;
} # html_quote();


sub error {
        my($message) = @_;

        print "--------------------\n";
        print "** Error Message **\n";
        print "--------------------\n\n";
        print $message . "\n";
        print "Author = J. Blustein\n";
        die;
} # error()


sub now {
        use POSIX qw(strftime);

        return strftime("%a %d %b %Y %X", localtime(time));
    } # now()
```

```
sub log_entry {
    my($OUT,$from,$to,$extra) = @_;

    print $OUT "\n" . &now . "\n";                                    120
    print $OUT " f='$from' t='$to'\n n='$extra'\n";
} # log_entry()
```

For my application it was important that only some people could read my documents, so I made the users fill in an HTML form which recorded their ID in a cookie and created a log file for them. After that, every time they followed a link the CGI program determined from the cookie who they were and appended the the appropriate log file. The code for processing the form is mostly straightforward (half of the instructions check for possible errors, such as invalid IDs or I/O troubles). After the log file has been set up however the program must send two messages to the server they must be sent together because the server can only receive one message. Figure 2.3 shows the relevant code from my form processing CGI program (which incidentally uses the aforementioned `CGI.pm`).

```
#!/usr/local/bin/perl -Tw

BEGIN {
   use CGI qw(:standard); # for cookie code
   $| = 1; # fflush after every print
}

{
   my $input = new CGI;

#...

   # – Cookie with multiple values
     #++
     # N.B.: It is safer to send $user and $art in a cookie than to send
     #   the log file name.  The log file name could be faked and not
     #   easily checked.  The article and session numbers could be faked
     #   but must still be within the valid range and cannot be used to
     #   overwrite existing logs.  Etc.
     #–
   # – Make cookie
   $mycookie = $input->cookie(-name => 'values',
                  # no expires value so cookie will stay until Mozilla exits
                  # or expiry is forced
                           -value  => {'user'=> $user,
                                       'etc' => $etc});

   # – Write cookie and redirect at the same time
   print $input->header(-type => 'Multipart/mixed',
                        -cookie  => $mycookie,
                        -Location => $redirect_to);
}
```

Figure 2.3: Listing of Part of Form Processing

Chapter 3

Summary

This article and the accompanying code have shown how you can track the links a user follows while reading files at your website. Regrettably there is no standard way to accomplish this in all browsers. I used a CGI cookie to identify the user, a CGI program (with its output hidden in a zero-size frame) to record which links were followed. A JavaScript program (written for Netscape's version of JS) enabled the CGI program to record the link selection at the same time and follow the link without reloading the entire document.

References

[1] Blustein, J.
Automatically generated hypertext versions of scholarly articles and their evaluation.
In *Proceedings of the Eleventh ACM Conference on Hypertext and Hypermedia* (San Antonio, TX, 30 May – 4 June 2000), ACM Press, pp. 201 – 210.

[2] Christiansen, T.
HTML hacking with regular expressions.
The Perl Journal 1, 1 (Spring 1996), 24 – 29.
⟨`URL:http://orwant.www.media.mit.edu/tpj/programs/Issue_1_HTMLregexps/findin%g_links_2`⟩.

[3] Nielsen, J.
The need for speed (alertbox for march 1997).
⟨`URL:http://www.useit.com/alertbox/9703a.html`⟩.

Appendix A

Simple Link Insertion Code

The program is not pretty but it does the job.

Program Code

```
#!/usr/bin/perl -w00

#++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++ ++
#
# Copyright (c) 1997-2002 James Blustein
#
# TITLE:        insert-link-track-calls.pl (insert-calls.pl for short)
# AUTHOR:       J. Blustein <jamie@cs.dal.ca> (see below for credit to
#                                              basic source)
# CREATED:      17 Nov 1997                                              10
# LAST MODIFIED: 15 Sep 2002 (v3.0.2a)
#
# PURPOSE: Replaces HTML HREF anchors (anchors with outgoing links) with
#          calls to the JavaScript function used with the link-tracker.pl
#          script to record which links readers follow.
#
# NOTES: The HTML source code must be valid (according to the HTML 3.2
#       standard or below) and abide by the following additional three
#       restrictions:
#         (a) no comments inside of anchor elements                      20
#         (b) no occurance of either '<A' or '<a' (two-character strings)
#             anywhere in the code except in anchors, including comments!
```

```
#         (c) no empty lines inside anchors
#
#       The HTML parsing might be more robust (and easier) with the HTML
#       module from CPAN.
#
##############################################################################
# Based on a program by Tom Christiansen:
#                                                                          30
#   HTML Hacking with Regular Expressions
#   by Tom Christiansen
#   in The Perl Journal
#      vol. 1, no. 1, Spring 1996
#      pages 24 - 29
#
# Original source obtained from
#   <URL:http://orwant.www.media.mit.edu/tpj/programs/Issue_1_HTMLregexps/
#        finding_links_2>
#   on 17 November 1997                                                    40
#
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

require 5.004;

use CGI qw(escape);
use strict;
use File::Basename;

use constant VERSION => 'insert-calls.pl 3.0.2a (02 Jun 1999)';           50
use constant DEBUG_OUTPUT => 0; # 0 is off
use constant VERBOSE => 1;
use constant CALL_CGI_DIRECTLY => 0; # call JavaScript iff 0,
                                     # call CGI otherwise

use constant JSPROG  => "go"; # go(HREF, NAME, TITLE)
 use constant CGIPROG => "/cgi-bin/tracker";
 use constant DOC     => "1";
 use constant CGI_SEP => "&";
                                                                          60
my($PROG) = basename($0);
```

```
BEGIN {
    print STDERR "This is `" . VERSION . "'\n";
}

#++
#  Constants
#    CGIPROG: The CGI program to call when a link is followed.
#             The NAME, HREF, and TITLE of the referring anchor will be
#             passed to the CGI program
#
#    DOC:     Something that can be resolved to the address (URL) of the
#             document into which the CGI calls are being inserted. This
#             is needed to resolve all the HREFs. You could use the entire
#             URL but that would take much longer to transmit than a short
#             identification number.
#
#    CGI_SEP: Character that separates arguments to CGI program.
#             According to the specification a semicolon or an ampersnad
#             can be used (with a preference for a semicolon). I have to
#             use ampersand because the CGI.pm module doesn't work with
#             semicolons.
#-


#++
# Global variables
#-
my(%unique_name); # these values are used below to create a name for links
                  # where no name is provided in the anchor

#- About %pattern and %tag -
#  For my experiment I made several types of hypertext links (all of which
# were coded in HTML). The NAME of the anchor included a label to identify
# the link type.

my %pattern; # identifying regexps for the major types of links
my %tag;     # if a link has no name then it will be assigned one
             # composed of its $tag and a unique number

$tag{sem}   = "S"; # S for semantic
```

```
$tag{defn}  = "D"; # D for definition
$tag{struct} = "P"; # P for plain
$tag{other} = "L"; # L for link

$unique_name{sem} = 1;
$unique_name{defn} = 1;
$unique_name{struct} = 1;
$unique_name{other} = 1;                                        110

$pattern{sem} = '(?:' . join ('|', qw{
                                    sent2sect
                                    s2sect
                                    sent2sent
                                    s2sent
                                    summary
                                    summ
                                    sent2para
                                    s2p                         120
                                    s2para
                                    wg2sent
                                   }
                              )
                    . ')';

$pattern{defn} = '(?:' . join ('|', qw{
                                     defn:sys
                                     defn:term
                                     defn:abbr                  130
                                     defn
                                    }
                               )
                     . ')';

$pattern{struct} = '(?:' . join ('|', qw{
                                       fn:back-jump
                                       list:LoF
                                       list:LoT
                                       list:ToC                 140
                                       list:ToS
                                       list:jump
```

```
                list:localC
                struct
                struct:BottomUp
                struct:Footnote
                struct:LoF
                struct:LoT
                struct:Ref
                struct:Up2ToC                                    150
                struct:manual
               }
        )
. ')';
```

```
# METHOD:
#  If there is an anchor in this paragraph then grab the text before and
#  after the anchor and the parts of the anchor itself (for processing in
#  the while loop).  Otherwise just print out the input line exactly as it
#  appears.

while (<>) {
  if ( m{<A} ) {
    while ( m{      # match repeatedly with /g
      \G(.*?)       # text before the anchor
                    #
      < \s* A       # this is an anchor
        \s+         #
        (HREF       # a link spec           \
        |NAME       # or a link destination + saved as $2
        |TITLE)     # or a link name       /
        \s* = \s*   # here comes the field described by $2
          ( ["'] )  # either quote, saved in $3 and \3
          ( .*? )   # the whole link, saved in $4
            \3      # the original $3 quote
                    #
        (?:\s*      # perhaps a second field
         (HREF      # a link spec        \
         |NAME      # or a link destination + saved as $5
         |TITLE)    # or a link name     /
         \s* = \s* # here comes the field described by $5
           ( ["'] ) # either quote, saved in $6 and \6
           ( .*? )  # the whole link, saved in $7
             \6     # the original $6 quote
        )?          # there might not be a second field
                    #
        (?:\s*      # perhaps a third field
         (HREF      # a link spec        \
         |NAME      # or a link destination + saved as $8
         |TITLE)    # or a link name     /
         \s* = \s* # here comes the field described by $8
           ( ["'] ) # either quote, saved in $9 and \9
           ( .*? )  # the whole link, saved in $10
             \9     # the original $9 quote
        )?          # there might not be a third field
```

```
                #
    (.*? >)     # the rest of the tag
                #
    (.*?)       # text after the tag up to                          200
    (?=<A|\Z)   # the end of the line or the next tag
        }xsgi)  # /x for expanded patterns
                # /s so . can match \n
                # /g to get multiple hits in one paragraph
                # /i for case insensitivity on A and HREF|NAME|TITLE
{
    my(@@field) = (); # a place to keep the patterns matched above
    my($after);  # text that appeared on the line but before the <A
    my($before); # text that appeared on the line but after the <A
    my($href);   # value of HREF field                               210
    my($name); # value of NAME field
    my($title);  # value of TITLE field
    my($rest);   # all of the other parts of the <A .. </A>

    #- copying the patterns matched into declared variables
     # Would split() do this better?
     # Would pointers help here?
     # This works and that's good enough for me now and it can be
     # improved in a future version
    $field[ 1] =  $1; $field[ 2] =  $2; $field[ 3] =  $3;           220
    $field[ 4] =  $4; $field[ 5] =  $5; $field[ 6] =  $6;
    $field[ 7] =  $7; $field[ 8] =  $8; $field[ 9] =  $9;
    $field[10] = $10; $field[11] = $11; $field[12] = $12;

    $before=$field[ 1] if defined $field[ 1];
    $rest  =$field[11] if defined $field[11];
    $after =$field[12] if defined $field[12];
```

```
#– filling in the values of $href, $name, and $title
#– from the patterns matched above                                    230
$href="";
$href=$field[ 4] if $field[ 2] =~ m/HREF/i;
$href=$field[ 7] if defined $field[ 5] && $field[ 5] =~ m/HREF/i;
$href=$field[10] if defined $field[ 8] && $field[ 8] =~ m/HREF/i;

$name="";
$name=$field[ 4] if $field[ 2] =~ m/NAME/i;
$name=$field[ 7] if defined $field[ 5] && $field[ 5] =~ m/NAME/i;
$name=$field[10] if defined $field[ 8] && $field[ 8] =~ m/NAME/i;
                                                                     240
$title="";
$title=$field[ 4] if $field[ 2] =~ m/TITLE/i;
$title=$field[ 7] if defined $field[ 5] && $field[ 5] =~ m/TITLE/i;
$title=$field[10] if defined $field[ 8] && $field[ 8] =~ m/TITLE/i;

# if there is an HREF then replace it with our CGI call
# else print out the anchor unchanged (except possibly reordered)
my($elements)="";
$elements .= "TITLE=\"$title\" " if "" ne $title;
$elements .= "NAME=\"$name\" " if "" ne $name;                       250

if ("" eq $href) {
    $elements .= "$rest" if "" ne $rest;
    print "$before<A $elements$after";
} else {
    my($cgi_args) = "";
    if ("" eq $name) { # every link needs a NAME, so when there
                       # isn't one provided we make one using a
                       # global count
                       # (this would be a great place for           260
                       #  an array & loop to make the program
                       #  easier to read and update.)
        if ($title =~ /^$pattern{sem}\s/) {
            $name = $tag{sem} . $unique_name{sem}++;
        }
        elsif ($title =~ /^$pattern{defn}$/) {
            $name = $tag{defn} . $unique_name{defn}++;
        }
```

```
            elsif ($title =~ /^$pattern{struct}$/) {
                $name = $tag{struct} . $unique_name{struct}++;
            }
            else {
                $name = $tag{other} . $unique_name{other}++;
            }
          $elements .= "NAME=\"$name\" ";
        }

        if (CALL_CGI_DIRECTLY) {
          $cgi_args .= "n=".&escape($name) .CGI_SEP; # coming from
                                                    # named anchor
                                                    # in ...
          $cgi_args .= "d=".DOC           .CGI_SEP; # this document
          $cgi_args .= "h=".&escape($href) .CGI_SEP; # to here, with
          $cgi_args .= "t=".&escape($title) .CGI_SEP; # this additional
                                                    # info attached
          $elements .= "HREF=\"" . CGIPROG . "?$cgi_args\" ";
          $elements .= "$rest" if "" ne $rest;
        } else { # inserting call to JavaScript instead
            $elements .= "onClick=\"" . JSPROG .
                      "('" . &escape($href) . "', '" .
                            &escape($name) . "', '" .
                              &escape($title) . "')\" ";
          $elements .= "HREF=\"" . $href . "\" ";
          $elements .= "$rest" if "" ne $rest;
        }
        print "$before<A $elements$after";
      }
    } # while we're in an anchor
  } else { # no anchors found in the input line
      print;
  }
}
```