

HW2 Web



Webviborous SiteSuckeri

Part A -- SiteSucker

For part A, you will build a little web client that sucks urls out of web pages.

Due midnight ending Thu May 9th. As before, P/NC students may work in teams of 2, and they are not required to do part B

The WebConnect class represents a connection to a single URL (see the starter file for details). A WebConnect object responds to a doConnect() message which causes it to make an HTTP/1.0 GET request for its URL, and store the results (if we did HTTP/1.1, we would have to deal with chunked responses -- blech). Java includes some built-in classes for HTTP connections, but we are not using them directly. I'd like you to write the HTTP connection code yourself, to get more experience with sockets and to get an insiders sense of how the protocol works.

The static main() function in WebConnect gives a simple command-line interface to the class. The "-u url" command takes a url, attempts the connection, and prints the resulting status line for that url followed by the HTTP response header itself if present..

```
-> java WebConnect -u http://www.stanford.edu/class/cs193i/  
http://www.stanford.edu/class/cs193i/ OK type:text/html time:244  
HTTP/1.1 200 OK  
Date: Wed, 01 May 2002 16:18:09 GMT  
Server: Stronghold/3.0 Apache/1.3.19 RedHat/3014c WebAuth/2.5 (Unix) mod_ssl/2.8.1  
OpenSSL/0.9.6 WebAuth/2.5 mod_fastcgi/2.2.10  
Connection: close  
Content-Type: text/html
```

```

-> java WebConnect -u http://maosucks.stanford.edu/
http://maosucks.stanford.edu/  ERR-CONNECT
-> java WebConnect -u mailto:spam@cs.stanford.edu
mailto:spam@cs.stanford.edu SKIP
-> java WebConnect -u http://www.stanford.edu/class/cs193i
http://www.stanford.edu/class/cs193i ERR-301 type:text/html; charset=iso-8859-1
time:149 location:http://www.stanford.edu/class/cs193i/
HTTP/1.1 301 Moved Permanently
Date: Wed, 01 May 2002 16:34:07 GMT
Server: Stronghold/3.0 Apache/1.3.19 RedHat/3014c WebAuth/2.5 (Unix) mod_ssl/2.8.1
OpenSSL/0.9.6 WebAuth/2.5 mod_fastcgi/2.2.10
Location: http://www.stanford.edu/class/cs193i/
Connection: close
Content-Type: text/html; charset=iso-8859-1

```

The status line is made of several parts, separated by spaces...

http://www.stanford.edu/class/cs193i/ OK type:text/html time:244

- First, the URL itself
- A result word summarizing the status code of the GET request. For a successful 200 connection: "OK". If the connection was not opened because the protocol is not HTTP: "SKIP". If the connection could not be made for any other reason (bad hostname, connection refused, ...): "ERR-CONNECT". For any other code, such as 301, append the code number to the string "ERR-", such as "ERR-301".
- If the HTTP header was retrieved successfully, and it included a content-type header, print the type like this "type:text/html".
- If the HTTP header (and possibly body) was retrieved successfully, print the elapsed time to make the connection and retrieve the content, in milliseconds like this "time:244". The method `System.currentTimeMillis()` returns the current time in milliseconds as a long.
- If the result code was in the 300 range, and the header included a location: field, include "location:*new-url*" at the end of the status.

If the "-a" option is specified instead of "-u", the HTTP body should be printed after the HTTP header, so long as it is a "text/*" MIME type. The text in the header and body should be converted to use "\n" line endings...

```

-> java WebConnect -a http://www.stanford.edu/class/cs193i/test/basic.html
http://www.stanford.edu/class/cs193i/test/basic.html OK type:text/html time:196
HTTP/1.1 200 OK
Date: Wed, 01 May 2002 16:45:43 GMT
Server: Stronghold/3.0 Apache/1.3.19 RedHat/3014c WebAuth/2.5 (Unix) mod_ssl/2.8.1
OpenSSL/0.9.6 WebAuth/2.5 mod_fastcgi/2.2.10
Connection: close
Content-Type: text/html

```

```
<html>
```

```
<head>
<title>Basic</title>
</head>
```

```
<body>
<h1>Basic</h1>
```

A few basic URLs

```
<ul>
<li><a href=http://www.theonion.com/>The Onion</a>
<li><A HREF=http://www.stanford.edu/class/cs193i/test/a.html>a absolute</A>
<li><a href = b.html>b relative</a> <li><a name=foo href=
"c.html">c relative</a>
<li><a href=mailto:spam-me@cs.stanford.edu>mailto</a>
<li><a href=jeffsad.jpg>Jeff Sad image</a>
<li><a href = "http://www.stanford.edu/class/cs193i" name=bar>missing </a>
<li><a href = http://maosucks.stanford.edu/>no such host</a>
<li><a href = "binky:some/unknown.protocol" >unknown protocol</a>
</ul>

</body>
</html>
```

doConnect()

The WebConnect object encapsulates a url, and ivars such as "type" and "body" which represent the status items above. All the ivars start out with default values -- the ints are -1 and the strings are all null. When the doConnect() happens, the GET process gets as far as it can, setting the ivars as it goes. We'll give doConnect() a "linear" design -- if it errs out at some point, the ivars that have not been set yet will be left with their default values.

1. Check that the protocol is "http". If not, do not do the connection. Set a flag so that the status string knows to say "SKIP".
2. Attempt to open the socket (start timing)
3. Make the GET (or HEAD) HTTP/1.0 request. Include a "connection:close" in the request, to make it clear that the we do not want a persistent connection.
4. Try to read the HTTP header
5. Parse the result code and type out of the HTTP header. If the process fails before here, the code will still be -1 (which translates to "ERR-CONNECT" in the status string), and the head and body will be null.
6. If we're doing a GET, and the code is 200, and the type starts with "text/", attempt to download the body text.
7. Close the connection and note the elapsed time.

Milestone

Get -u and -a modes working. Most of the work is in doConnect() to perform the connection, and getStatus() to create the status string from the various ivars.

-x Mode

With the -x "extract" command "-x url", WebConnect should do a connection for the url header and body as above. If the body is present and is text/html, then extract all the urls from the body, do the relative-absolute conversion, and print them one per line...

```
-> java WebConnect -x http://www.stanford.edu/class/cs193i/test/basic.html
http://www.stanford.edu/class/cs193i/test/basic.html OK type:text/html time:723
http://www.theonion.com/
http://www.stanford.edu/class/cs193i/test/b.html
http://www.stanford.edu/class/cs193i/test/c.html
mailto:spam-me@cs.stanford.edu
http://www.stanford.edu/class/cs193i/test/jeffsad.jpg
http://www.stanford.edu/class/cs193i
http://maosucks.stanford.edu/
```

Parsing

There are some methods in ParseUtil.java to help with the parsing problem -- see the extractHrefs() method. Href urls may look like...

 -- simple

 -- with quotes

 -- with other bindings
in the tag

<a name=
foo href=
bar.html binkymode=true> -- as above, but with more whitespace

<anthrax mode=true> -- looks like an <a ..>, but it's not

As a practical matter, hrefs may contain any character except:
whitespace, ", <, or >.

We will not do error checking on the HTML -- your code only needs to work for syntactically correct HTML and hrefs. Also, you do not need to do the right thing for hrefs inside HTML comments <!-- --> or weird cases where markup is hidden inside a quoted string. Fixing those cases is not that hard, but it's not that interesting either. Just make sure your parsing works for the URLs in the pages at <http://www.stanford.edu/class/cs193i/test/>

Parsing Strategy

Here's our recommended parsing strategy...

1. Find the index of a "<a" and the ">" that follows it. Pull out the tag text from between those two and work on it separately
2. Use a `StringTokenizer(string, "\r\t\f\n=\\"", true)`; -- treats all whitespace, =, and quotes (") as tokens. Use the tokenizer to work through the tag text left-right, ignoring tokens which are whitespace, looking for the sequence: `href, =, something`. If the something is a quote ("), then the url is everything up to the next quote. Otherwise the something is the url already.

Milestone

Get the parsing working so that `-x` prints out all the extracted hrefs in their raw, relative form first. The URL conversion can be added last, since it's easy.

URL Conversion

We'll let the built-in URL object do the relative-absolute conversion for us -- just use the ctor `URL(base-url, relative-url-string)`. If the protocol is not known (such as the binky: protocol above), the URL conversion will throw a `MalformedURLException`. In that case, we'll just ignore that URL and not print it in the `-x` listing. You may assume that the initial command-line URL is always an HTTP url, so that URL ctor never fails.

-c Mode

Finally, `-c "check"` mode is similar to `-x` mode, except it constructs a `WebConnect` for each nested URL, attempts a HEAD connection for each, and prints its status string. We do not download the content of each URL, we just do a HEAD to check that the URL is good...

```
-> java WebConnect -c http://www.stanford.edu/class/cs193i/test/basic.html
http://www.stanford.edu/class/cs193i/test/basic.html OK type:text/html time:221
http://www.theonion.com/ OK type:text/html time:145
http://www.stanford.edu/class/cs193i/test/b.html OK type:text/html time:31
http://www.stanford.edu/class/cs193i/test/c.html OK type:text/html time:23
mailto:spam-me@cs.stanford.edu SKIP
http://www.stanford.edu/class/cs193i/test/jeffsad.jpg OK type:image/jpeg time:19
http://www.stanford.edu/class/cs193i ERR-301 type:text/html; charset=iso-8859-1
time:12 location:http://www.stanford.edu/class/cs193i/
http://maosucks.stanford.edu/ ERR-CONNECT
```

This mode is slower, since it does many more network connections (using HEAD vs. GET compensates somewhat). You can use `-c` mode to try your solution out with real web-sites, although you may annoy them with the web traffic, and in some cases their URLs may fall outside of our assumptions. For our testing, we will use pages like the ones at the `cs193i/test` url above.

Part B -- WebBounce

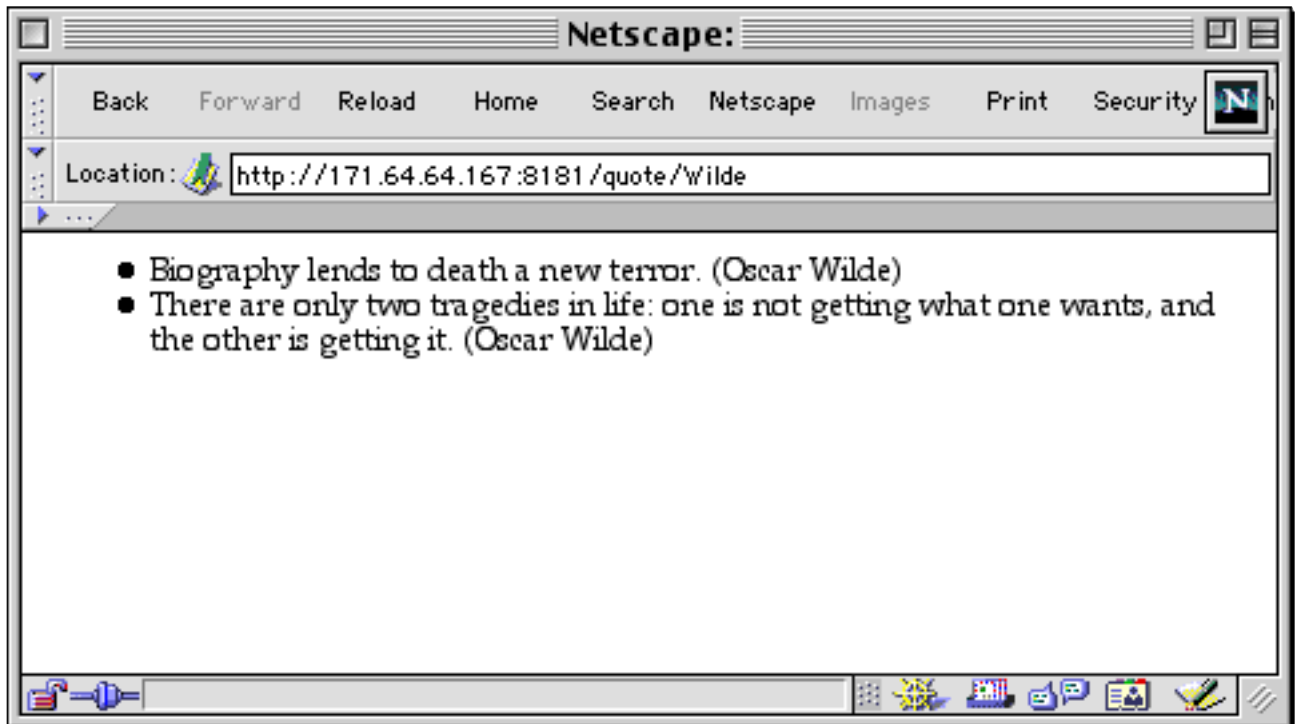
For Part B, you will build a little web server. Part B is smaller than Part A, but it's more devious. S/NC students do not need to do part B.

See the starter code in WebBounce.java. In its main() WebBounce should accept the port number to use and the quotes file ...

```
-> java WebBounce 8181 quotes.txt
```

WebBounce will deal with connections one at a time...

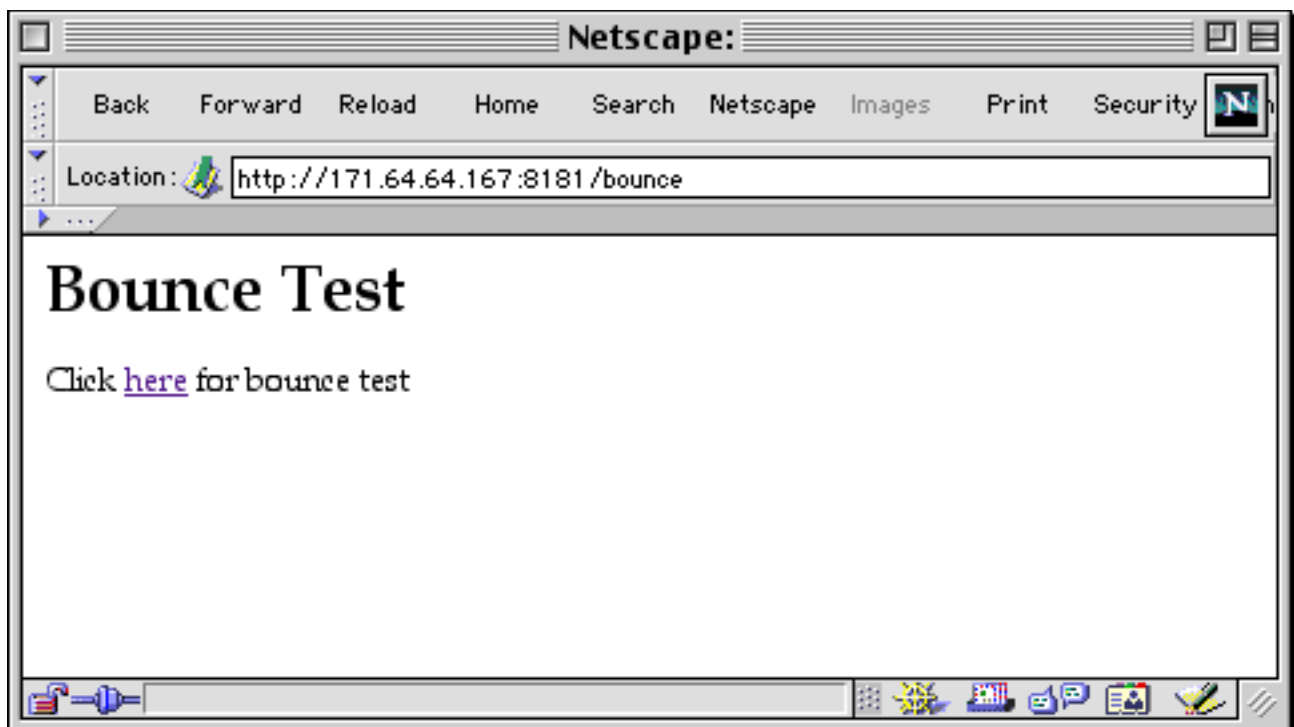
- Load the quotes file into an array, one line per element, at startup time.
- WebBounce will just look at the requested path in the first line of the client request.
- If the path is `"/quote/word"`, then, return a simple HTML page made of a ``, with one `` for each quote that included the given word as a substring (case-sensitive is ok). The returned header can be very simple: just the first HTTP response line and `content-type: text/html`.

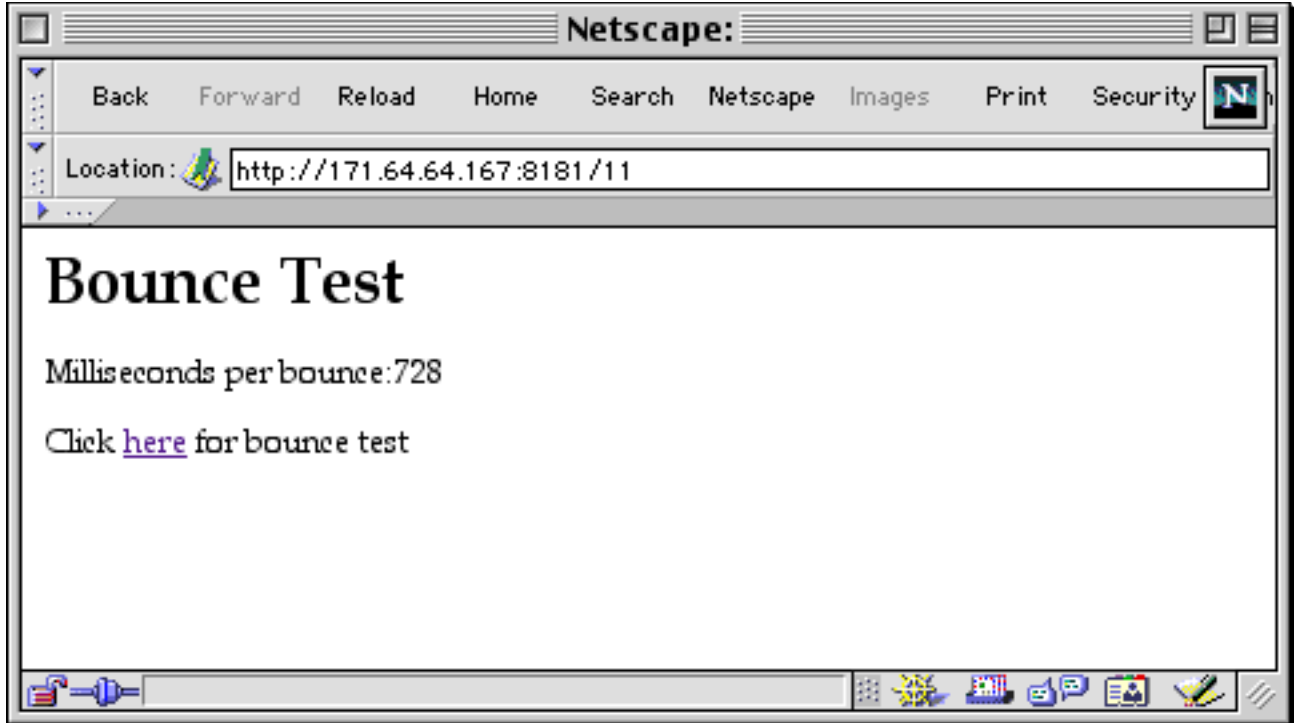


- If the path is "/bounce", then return the bounce starter page (below)
- All non-recognized requests can return 404.

Bounce Feature

The bounce feature is a trick to make the client do 10 requests as quickly as possible to measure the client/server latency. The bounce starter page is just plain HTML with a link: Click [here](#) for bounce test. The link href should be "/1". When the server gets the /1, it should note the time, and give a 301 redirect to location: /2, When getting the /2 request, the server should redirect to /3, and so on. When the server gets a /11, it should note the elapsed time, and return a result page: "Milliseconds per bounce: 34 Click [here](#) for bounce test." (divide the elapsed time by 10 to get "per bounce" time).





Cache Flow

When you run the test a second time, the browser may cache what it thinks the result is for "/1", which completely messes things up. Therefore, put a "cache-control: no-cache" in the response header.

Continuity

We'll assume that the "/1", "/2", ... "/11" series is from one client. If multiple clients try to run the test at the same time, we'll allow the timing to get messed up (we'll fix this "continuity" problem of HTTP in later homeworks). The time shown above is for a very slow setup -- hopefully your result will be more in the range of 50-100 milliseconds per bounce.

Absolute Location

It's convenient to think of the url in the redirect as being relative, such as "/1", but in reality it should be an absolute URL like "http://171.64.64.250:8181/1". Relative redirects will work in most browsers, but they're not following the RFC. Use `sock.getLocalAddress().getHostAddress()` on the client socket to get the "171.64.64.250" string, so you can build the proper absolute URL. We use the IP address instead of the DNS name to avoid the DNS lookup which might mess up the timing.