

The Wayback Machine - <https://web.archive.org/web/19981203153836/http://fishcam.netscape.com:80/assist/...>



AN EXPLORATION OF DYNAMIC DOCUMENTS

INTRODUCTION

This document explores the "server push" and "client pull" dynamic document capabilities of Netscape Navigator 1.1 (Windows, Mac, and Unix versions). Please send comments to pushpull@netscape.com. Also, if you are using either server push or client pull in a real-world application, please drop us a note at pushpull@netscape.com and let us know -- thanks!

Examples are given at the end, along with an important note on implementing server push CGI programs as shell scripts.

THE GREAT IDEA

The general idea is that browsers have always been driven by user input. You click on a link or an icon or an image and some data comes to you. As soon as people saw they could do that, they wanted to give a server the ability to push new data down to the browser. (An obvious example is a stock trader who wants to see new quote data every 5 minutes.) Up until now, that hasn't been possible.

Netscape Navigator 1.1 gives content creators and server administrators two new open standards-based mechanisms for making this work. The mechanisms are similar in nature and effect, but complementary. They are:

1. **Server push** -- the server sends down a chunk of data; the browser display the data but leaves the connection open; whenever the server wants it sends more data and the browser displays it, leaving the connection open; at some later time the server sends down yet more data and the browser displays it; etc.
2. **Client pull** -- the server sends down a chunk of data, including a directive (in the HTTP response or the document header) that says "reload this data in 5 seconds", or "go load this other URL in 10 seconds". After the specified amount of time has elapsed, the client does what it was told -- either reloading the current data or getting new data.

In server push, a HTTP connection is held open for an indefinite period of time (until the server knows it is done sending data to the client and sends a terminator, or until the client interrupts the connection). In client pull, HTTP connections are never held open; rather, the client is told when to open a new connection, and what data to fetch when it does so.

In server push, the magic is accomplished by using a variant of the MIME message format "multipart/mixed", which lets a single message (or HTTP response) contain many data items. In client pull, the magic is accomplished by an HTTP response header (or equivalent HTML tag) that tells the client what to do after some specified time delay.

We'll explore client pull first...

CLIENT PULL

FreeWheel, Ex. 1079
FreeWheel v. AlmondNet
IPR2024-00419

A simple use of client pull is to cause a document to be automatically reloaded on a regular basis. For example, name the following document "doc1.html" and try loading it in Netscape Navigator 1.1:

```
<META HTTP-EQUIV="Refresh" CONTENT=1>
<title>Document ONE</title>

<h1>This is Document ONE!</h1>

Here's some text. <p>
```

You will notice that the document reloads itself once a second.

What's happening? Simply put, we're using the `META` tag (a standard HTML 3.0 tag, for simulating HTTP response headers in HTML documents) to tell the browser that it should pretend that the HTTP response when "doc1.html" was loaded included the following header:

Refresh: 1

That HTTP header, in turn, tells the browser to go reload (refresh) this document after 1 second has elapsed. If we wanted to wait 12 seconds instead, we could have used this HTML directive:

```
<META HTTP-EQUIV="Refresh" CONTENT=12>
```

...which is equivalent to this HTTP response header:

Refresh: 12

Note: You should make sure the `META` tag is used inside the `HEAD` of your HTML document. That means it must appear before any text or images that would be displayed as part of your document.

So that's pretty cool.

A couple things to notice:

- In this example, a new "Refresh" directive (via either the `META` tag or the Refresh HTTP response header) is given as a part of every retrieval. This is an important point. **Each individual "Refresh" directive is one-shot and non-repeating. The directive doesn't say "go get this page every 6 seconds from now until infinity"; it says "go get this page in 6 seconds".**

If you want continuous reloading, you have to give the directive on each retrieval. If you only want the document reloaded once, only give the directive on the first retrieval.

- Once given the directive, the browser *will* do the specified retrieval after the specified amount of time. The only way to cause it not to happen is to not have an open window that contains the document.

This also means that if you set up an "infinite reload" situation (as the example above does), the only way you can interrupt the infinite reloading is by pressing the "Back" button or otherwise going to a different URL in the current window (or, equivalently, by closing the current window).

So another thing you obviously want to do, in addition to causing the current document to reload, is to cause another document to be reloaded in n seconds in place of the current document. This is easy. The HTTP response header will look like this:

```
Refresh: 12; URL=http://foo.bar/blatz.html
```

The corresponding META tag would be:

```
<META HTTP-EQUIV="Refresh" CONTENT="12; URL=http://foo.bar/blatz.html">
```

Important note: make sure the URL you give is fully qualified (e.g. <http://whatever/whatever>). That is, don't use a relative URL.

Here are two example documents, "doc2.html" and "doc3.html", each of which causes the other to load (so if you load one of them, your browser will flip back and forth between them indefinitely). Here's "doc2.html":

```
<META HTTP-EQUIV=REFRESH CONTENT="1; URL=http://machine/doc3.html">
<title>Document TWO</title>
```

```
<h1>This is Document TWO!</h1>
```

```
Here's some other text. <p>
```

```
<META HTTP-EQUIV=REFRESH CONTENT="1; URL=http://machine/doc2.html">
<title>Document THREE</title>
```

```
<h1>This is Document THREE!</h1>
```

```
Here's yet more text. <p>
```

(You should tweak the URLs to match your local system. Remember, they must be fully specified.)

Now load one of the documents; the browser will load the other in 1 second, then the first in another second, then the second again in another second, and so on forever.

How do you make it stop? The easiest way is to either close the window, or put a link in the document(s) that points to somewhere else. Remember, any retrieval of any document can cause the whole process to stop at any point in time if a fresh directive isn't issued -- the process only continues as long as each new document causes it to continue. Thus, the content creator has total control.

Neat trick: the interval can be 0 seconds! This will cause the browser to load the new data as soon as it possibly can (after the current data is fully displayed).

Another neat trick: the data that is retrieved can be of any type: an image, an audio clip, whatever. One fun thing to envision is 0-second continuous updating of a live image (e.g. a camera feed), or a series of still images. Poor man's animation, kind of. We're considering mounting a camouflaged IndyCam on the prow of Jim Clark's boat and feeding live images to the world using this mechanism.

Yet another neat trick: a "Refresh" header can be returned as part of any HTTP response, including a redirection. So a single HTTP response can say "go get this URL now, and then go get this other

URL in 10 seconds".

This means you can have a continuous random URL generator! Have a normal random URL generator (such as [URouLette](#)) that returns as part of its redirection response a "Refresh" directive that causes the browser to go get another random URL from the random URL generator in 18 seconds.

SERVER PUSH

Server push is the other dynamic document mechanism, complementing client pull.

In contrast to client pull, server push takes advantage of a connection that's held open over multiple responses, so the server can send down more data any time it wants. The obvious major advantage is that the server has total control over when and how often new data is sent down. Also, this method can be more efficient, since new HTTP connections don't have to be opened all the time. The downside is that the open connection consumes a resource on the server side while it's open (only when the server knows it wants this to happen, though). Also, server push has two other advantages: one is that a server push is easily interruptible (you can just hit "Stop" and interrupt the connection), and the other advantage we'll talk about a little later.

First, a short review: the MIME message format is used by HTTP to encapsulate data returned from a server in response to a request. Typically, an HTTP response consists of only a single piece of data. However, MIME has a standard facility for representing many pieces of data in a single message (or HTTP response). This facility uses a standard MIME type called "multipart/mixed"; a multipart/mixed message looks something like:

```
Content-type: multipart/mixed;boundary=ThisRandomString
```

```
--ThisRandomString
Content-type: text/plain
```

```
Data for the first object.
```

```
--ThisRandomString
Content-type: text/plain
```

```
Data for the second and last object.
```

```
--ThisRandomString--
```

The above message contains two data blocks, both of type "text/plain". The final two dashes after the last occurrence of "ThisRandomString" indicate that the message is over; there is no more data.

For server push we use a variant of "multipart/mixed" called "multipart/x-mixed-replace". The "x-" indicates this type is experimental. The "replace" indicates that each new data block will cause the previous data block to be replaced -- that is, new data will be displayed **instead of** (not in addition to) old data.

So here's an example of "multipart/x-mixed-replace" in action:

```
Content-type: multipart/x-mixed-replace;boundary=ThisRandomString
```

```
--ThisRandomString
Content-type: text/plain
```

```
Data for the first object.
```

```
--ThisRandomString
Content-type: text/plain
```

Data for the second and last object.

```
--ThisRandomString--
```

The key to the use of this technique is that the server does not push the whole "multipart/x-mixed-replace" message down all at once but rather **sends down each successive data block whenever it sees fit**. The HTTP connection stays open all the time, and the server pushes down new data blocks as rapidly or as infrequently as it wants, and in between data blocks the browser simply sits and waits for more data in the current window. The user can even go off and do other things in other windows; when the server has more data to send, it just pushes another data block down the pipe, and the appropriate window updates itself.

So here's exactly what happens:

- Following in the tradition of the standard "multipart/mixed", "multipart/x-mixed-replace" messages are composed using a unique boundary line that separates each data object. Each data object has its own headers, allowing for an object-specific content type and other information to be specified.
- The specific behavior of "multipart/x-mixed-replace" is that each new data object replaces the previous data object. The browser gets rid of the first data object and instead displays the second data object.
- A "multipart/x-mixed-replace" message doesn't have to end! That is, the server can just keep the connection open forever and send down as many new data objects as it wants. The process will then terminate if the user is no longer displaying that data stream in a browser window or if the browser severs the connection (e.g. the user presses the "Stop" button). We expect this will be the typical way people will use server push.
- The previous document will be cleared and the browser will begin displaying the next document when the "Content-type" header is found, or at the end of the headers otherwise, for a new data block.
- The current data block (document) is considered finished when the next message boundary is found.
- Together, the above two items mean that the server should push down the pipe: a set of headers (most likely including "Content-type"), the data itself, and a separator (message boundary). When the browser sees the separator, it knows to sit still and wait indefinitely for the next data block to arrive.

Putting it all together, here's a Unix shell script that will cause the browser to display a new listing of processes running on a server every 5 seconds:

```
#!/bin/sh
echo "HTTP/1.0 200"
echo "Content-type: multipart/x-mixed-replace;boundary=---ThisRandomString---"
echo ""
echo "---ThisRandomString---"
while true
do
echo "Content-type: text/html"
echo ""
echo "<h2>Processes on this machine updated every 5 seconds</h2>"
echo "time: "
date
echo "<p>"
echo "<plaintext>"
ps -el
echo "---ThisRandomString---"
```

sleep 5
done

Note that the boundary is sent to the browser before the sleep statement. This ensures that the browser will flush its buffers and display all the data that's been received up to that point to the user.

Special note to NCSA HTTPD users: You must not use any spaces in your content type, this includes the boundary argument. NCSA HTTPD will only accept a single string with no white space as a content type. If you put any spaces in the line (besides the one right after the colon) any text after the white space will be truncated.

As an example, the following will work:

```
Content-type: multipart/x-mixed-replace;boundary=ThisRandomString
```

The following will not work:

```
Content-type: multipart/x-mixed-replace; boundary=ThisRandomString
```

THE AFOREMENTIONED OTHER ADVANTAGE TO SERVER PUSH

You can use server push for individual inlined images! Yes, that's right -- you can have a document that contains an image that gets updated by the server on a regular basis or any time the server wants. Just have the SRC attribute of the IMG tag point to an URL for which the server pushes a series of images.

Let's stress this point: if you use server push for an individual inlined image, the image will get replaced inside the document each time a new image is pushed -- the document itself won't get touched (assuming it isn't separately subject to server push).

So this is kind of cool -- poor man's animation inlined into a static document.

EFFICIENCY CONSIDERATION

Server push: generally more efficient, since a new connection doesn't need to be opened for each new piece of data. Since a connection is held open over time, even when no data is being transferred, the server must be willing to accept dedicated allocation of a TCP/IP port, which may be an issue for servers with a sharply limited number of TCP/IP ports.

Client pull: generally less efficient, since a new connection must be opened for each new piece of data. However, no connection is held open over time.

Note that in real world situations it is common for establishment of a new connection to take a significant amount of time -- i.e., one second or more. Given that this is the case, server push will probably be generally preferable for end-user performance reasons, particularly for information that is frequently updated.

Another consideration is that the server has comparatively more control in the server push situation than in the client pull situation. One example: there is one distinct open connection for each instance of server push in use, and the server can elect to (for example) shut down such a connection at any time (e.g., via a cron daemon) without requiring a whole lot of logic in the server. On the other hand, the same application using client pull will look like many independent connections to the server, and the server may need to have a considerable level of complexity in order to manage the situation (e.g., associating client pull requests with particular end users to figure out who to stop sending new "Refresh" headers to).

An Important Note On Server Push And Shell Scripts: If you write a CGI program as a shell script, and the script implements some form of server push where you expect the connection to be open for a long time (e.g. an infinitely long stream of images representing live video), then the shell script normally will not notice when/if the user severs the connection on the client side (e.g., by pressing the "Stop" button) and will continue running. This is bad, as server resources will be thereafter consumed wastefully and uselessly. The easiest way to work around this shell script limitation is to implement such CGI programs using a language like Perl or C -- such programs will terminate properly when the user breaks the connection.

AN EXAMPLE

[Mozilla icon animation](#) -- an inlined animation in a static document, via server push.

[Help](#) | [Site Map](#) | [Company Information](#) | [How to Get Netscape Products](#) | [Advertise With Us](#) | [Add Site](#)
[Autos](#) | [Business](#) | [Computing & Internet](#) | [Education](#) | [Entertainment](#) | [Games](#) | [Health](#) | [Kids & Family](#) | [Lifestyles](#) |
[Local](#) | [Netscape](#) | [News](#) | [Personal Finance](#) | [Real Estate](#) | [Shopping](#) | [Small Business](#) | [Sports](#) | [Travel](#)

© 1998 Netscape, All Rights Reserved. [Legal & Privacy Notices](#)
This site powered by [Netscape SuiteSpot servers](#).