A network file system over HTTP: remote access and modification of files and *files*

Oleg Kiselyov

oleg@pobox.com oleg@computer.org oleg@acm.org
http://pobox.com/~oleg/ftp/USENIX99/

Abstract

The goal of the present HTTPFS project is to enable access to remote files, directories, and other containers through an HTTP pipe. HTTPFS system permits retrieval, creation and modification of these resources as if they were regular files and directories on a local filesystem. The remote host can be *any* UNIX or Win9x/WinNT box that is capable of running a Perl CGI script and accessible either directly or via a web proxy or a gateway. HTTPFS runs entirely in user space. The current implementation fully supports reading as well as creating, writing, appending, and truncating of files on a remote HTTP host. HTTPFS provides an isolation level for concurrent file access stronger than the one mandated by POSIX file system semantics, closer to that of AFS. Both a programmatic interface with familiar open(), read(), write(), close(), etc. calls, and an interactive interface, via the popular Midnight Commander file browser, are provided.

Overview

Unlike NFS and AFS, HTTP is supported on nearly all platforms, from IBM mainframes to PalmPilots and cellular phones, with a widely deployed infrastructure of proxies, gateways, and caches. It is also regularly routed through firewalls. Using standard HTTP `GET`, `PUT`, `HEAD` and `DELETE` request methods, a rudimentary network file system can be created that runs cross-platform (e.g., Linux, Solaris, HP-UX, and Windows NT) on a variety of off-the-shelf HTTP servers: Apache, Netscape, and IIS. The HTTPFS can be used either programmatically or via an interactive interface.

HTTPFS is a user-level file system, implemented by a C++ class library on a client site, and a Perl CGI script on a remote site. The C++ framework of `VNode`, `VNode_list`, `HTTPTransaction`, `MIMEDiscreteEntity` etc. classes may be employed directly. Alternatively, HTTPFS functionality can be extended to arbitrary applications by linking with a library that transparently replaces standard file system calls (e.g., open(), stat(), and close()). This operation does not patch the kernel or system libraries, nor does it require system administrator privileges. The interposed functions invoke the default implementations, unless a file with an "`http://`" prefix is accessed. The HTTPFS client framework will handle the latter case. This permits URLs being used whenever a regular file name is expected, as an argument to `open()`, `fopen()`, `fstream()`, or a command-line parameter to file utilities. No source code needs to be modified, or even recompiled.

An important feature of HTTPFS is that it can provide a file-centric view of remote resources and containers that are not necessarily files or directories on a remote computer. Anything which an HTTPFS server can apply `GET`, `PUT`, `DELETE` methods to, and has timestamps and size attributes, may be accessed and manipulated as if it were a file. With HTTPFS, an off-the-shelf application may open(), read(), write() a "file" that may in reality be a database table, an element in an XML document, a property in the registry, an ARP cache entry, or the input or output of a process.

Borrowing from database terminology, HTTPFS provides an isolation level of "Repeatable Read" for concurrent file transactions. Once a process opens a file, it will not see changes to the file made by other concurrently running processes. This isolation is different from standard POSIX semantics, which provides for a "Dirty Read" isolation – updates made to the file by other processes are visible before the file is closed. The difference in semantics is important, but only when a file is being concurrently read and modified. As was mentioned above, HTTPFS may permit a file-type access to a table of a relational database. In this particular case, the "Repeatable Read" isolation level is appropriate as it is the default for an ANSI-compliant database.

Hypertext Transfer Protocol

HTTP is an application-level protocol for distributed, collaborative, hypermedia information systems [1]. It is a request/response protocol, where the client submits a request to the server, the server processes the request, and sends a response to the client. HTTP is open-ended, in that it allows new request/ response pairs to be defined. The message format is similar to that used by Multipurpose Internet Mail Extensions (MIME).

An HTTP transaction is in some sense a remote procedure call. An HTTP message specifies both an operation and the data on which to invoke the operation. The protocol provides facilities for exchanging data (arguments and results), and meta-data. The latter specialize a request and a response, carry authentication information and credentials, or annotate the content. Most HTTP transactions are synchronous, although HTTP/1.1 provides for asynchronous and batch modes. Furthermore, HTTP allows intermediaries (caches, proxies) to be inserted into the response-reply chain.

An HTTP request includes the name of the operation to apply and the name of the resource. Additional parameters if needed are communicated via request headers, or a request body. The request body may be an arbitrary stream of bytes. The HTTP/1.1 standard defines methods `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `OPTIONS`, and `TRACE`, which can be further extended by a particular server.

• The `GET` method retrieves the requested data along with some meta-information about the data.The data is denoted by a URI (universal resource identifier). The `GET` method can be conditional; if the resource has not been modified since the specified date, no data is returned. This form is useful when a cached copy of the resource exists.

• The `HEAD` method works similarly to the `GET` method, except that the server returns only the meta-data describing the properties of a resource.

• The `PUT` method stores the supplied data in the specified URI. Once `PUT,` the data will be available via a later `GET`.

HTTPFS maps these methods to the corresponding file access operations, while fully preserving the methods' semantics defined in the HTTP/1.1 document.

Of particular interest is the extensibility of the HTTP protocol. A client can submit arbitrary headers, which are available to the corresponding web server. The server may send arbitrary meta-data as response headers as well. In addition, a client and a server may exchange meta-information via "name=value" attribute pairs of the standard `Content-Type:` header.

Implementation of HTTPFS: Client

HTTPFS is implemented by a C++ framework. It carries out HTTP transactions with a server and maintains a local cache of fetched files and directory listings. A file being opened for reading or modification is first fetched from a server in a `GET` transaction. However, if the file is already in cache, a conditional `GET` request is issued to verify that the cached copy is up-to-date, and reload it if not. When a file is being opened for writing, an additional `Pragma:` header is included in the `GET` request to inform the server of the open mode: `O_RDWR`, `O_WRONLY`, `O_CREAT`, `O_EXCL`, `O_TRUNC` or `O_APPEND`. The server may then create, truncate, or lock the resource. A response from the server is translated into the result of the `open()` call. Reads and writes to the opened file are then directed to the local copy. On `close()`, if the local copy has been modified, it is written back using `PUT`.

Status inquiries, e.g., `stat()`, `lstat()`, `readlink()`, etc., are implemented by submitting a `HEAD` request. A `Pragma:` request header tells the server which particular status information about the resource is requested.

Scanning of a directory – `opendir()`, `readdir()`, `closedir()` – is similar to accessing a file: a `GET` request is issued for a directory URI, and the resulting directory listing is locally cached.

Appendix A gives a detailed mapping between the file system API and HTTP requests and responses.

Implementation of HTTPFS: Server

A MCHFS server is one particular HTTPFS server. It is a Perl CGI script which executes HTTPFS requests and provides access to resources and containers. In the case of MCHFS, the resources and containers happen to be regular files and directories of a computer that runs this CGI script. The script thus lists directories on its own server, sends files, and accepts new content for old or newly created files.

According to a tradition, an HTTP server operates in a "`chroot`"ed environment. For example, when asked to retrieve a resource `http://hostname/README.html`, the server sends a file located at `$DOCUMENT_ROOT/README.html` (if exists), where `$DOCUMENT_ROOT` is something like `/opt/apache/htdocs`. MCHFS honors this convention:

```
open("http://hostname/cgi-bin/admin/MCHFS-server.pl/README.html", O_RDONLY)
```

will let you access the same `$DOCUMENT_ROOT/README.html` file. Still MCHFS offers to escape the "`chroot`"ed confines and access files anywhere in its file system. This can be accomplished by using a distinguished path component `DeepestRoot`, which refers to the root of the server's file system. For example:

```
open("http://hostname/cgi-bin/admin/MCHFS-server.pl/DeepestRoot/etc/passwd",O_RDONLY);
open("http://hostname/cgi-bin/admin/MCHFS-server.pl/DeepestRoot/WinNT/Profiles/Administrator/NTusers.dat",O_RDONLY);
```

This is discussed further in the section on security considerations, below.

MCHFS allows any web browser to view directory listings and files. A directory request is returned as plain text, in a format similar to a '`ls -l`' listing. Because MCHFS understands regular `GET` requests, you can use a web browser to verify that MCHFS is installed and functioning properly. Any other user agent – `Wget` or the plain `telnet` – may be employed as well.

Transparent replacement of system calls

An application accesses the file system API either using low-level open/read/write/close calls, or via abstract file system interfaces (e.g., standard I/O, stream I/O, or ports). The latter are implemented, under the covers, through the open/read/write/close. Once these low-level functions are impersonated (and extended to handle `http://` "file names"), HTTPFS becomes available to any application without modifying the application's source code.

One does not need to patch the kernel or system libraries to intercept the POSIX filesystem API calls. One can do it safely, and without system administrator privileges by linking the application with replacement versions of these low-level API functions. The recipe for doing so is as follows:

- compile a stub function with the name of the replaced routine;
- partially and statically link the stub with default implementations of the functions being intercepted;
- link the result with an application, the HTTPFS client library, and necessary standard libraries; the link mode may be either static, dynamic, or mixed.

Source code for an application is not required, only its object (compiled) form; the application need not be aware that it is using HTTPFS.

A web page [2] explains this technique in detail, and discusses another use of this interception approach: implementing processes-as-files.

HTTPFS and Midnight Commander

The Midnight Commander is a directory browser/file manager for Unix-like operating systems [3]. Its interface is similar to that of John Socha's Norton Commander for DOS as well as to Microsoft Windows' Explorer. The Midnight Commander (MC) can show the contents of two directories at the same time. Besides the file names, the views may display size, type, modification date, and other file attributes. The MC lets you select a group of files from the current view, and perform a number of operations (copy, rename, view, edit, etc) on the current file or selection with one or few keystrokes or mouse clicks.

MC supports remote file access via MCFS, a remote file access protocol that requires that an MCFS server be running on a remote machine. I provide an "adapter" – an MCFS server linked to a HTTPFS client – that translates MCFS orders to HTTPFS requests. Thus Midnight Commander gains an ability to access remote files via HTTPFS for free, without any modifications to its code.

The following sample session hopefully shows what good the HTTPFS/MC alliance can do. This is a transcript of an actual session, with only hostnames changed.

- `mc -d mc:sol-server/sol-server:80/cgi-bin/admin/MCHFS-server.pl/DeepestRoot/tmp`

This command launches MC and has it display the listing of a `/tmp` directory on a remote computer `sol-server` (SunSparc/Solaris 2.6). The `mc:sol-server` component in the "directory name" above refers to the computer that executes the MC/HTTPFS adapter. The adapter may run on the same computer with MC, or alongside the HTTPFS server, or on some other site.

- Select a file on the current pane, and press `F3`. A built-in MC viewer shows the contents of that remote file.
- With the selection bar still on that file, press `F4`. A built-in editor is launched, which lets you alter the remote file as if it were a local file.
- Press `F5` to copy the file to a local directory listed on the other MC's pane.
- Switch to that pane and type

`cd mc:sol-server/winnt-server/cgi-bin/admin/MCHFS-server.pl/wwwroot`

The pane lists a remote directory `DocumentRoot/wwwroot` on a WinNT host `winnt-server`, which runs IIS. The first MC pane still shows the contents of the `/tmp` directory on `sol-server`. By selecting files and pressing `F5`, you may copy files from one remote directory onto the other. In this example, MC, MC/HTTPFS adapter, and HTTPFS server are running on three different computers.

• You notice a file `US98talk.tar.gz` in the `sol-server:/tmp` directory. If you highlight the file and press `F3`, you can navigate this *remote* tar archive as if it were a directory tree. You can select files (members of that remote archive), view and copy them as if they were on your local filesystem.

Pushing the envelope and security holes

The MCHFS script obviously opens up the file system of a host computer to the entire world. Furthermore, if a particular HTTPFS server chooses to interpret `GET`/`PUT` requests as output/input from an application (`sh` in particular), the whole system becomes exposed. Clearly this may not be desirable. Therefore, one may want to restrict access to MCHFS to trusted hosts or users. These authentication/ authorization policies are the responsibility of a web server's administrator; MCHFS need not be aware of them.

In addition, the MCHFS server may implement its own resource restriction policies. For example, it can refuse `PUT` requests, which effectively makes exported file systems read-only. MCHFS could permit modification or listing of only certain files, or disallow use of `DeepestRoot` and "`..`" in file paths, thus confining users to a limited part of the file system tree.

Related work

HTTPFS is similar to FTPFS, a virtual file system used by Midnight Commander, Emacs and KDE to access remote FTP sites. There is also a similarity to NFS. There are, however, a number of differences:

• HTTPFS operates through TCP channels using HTTP, a simple stateless reliable protocol. HTTP is less resource-hungry than FTP.

• HTTPFS can talk to *any* host that runs an HTTP server and capable of executing a Perl CGI script.

• HTTPFS works transparently through firewalls, HTTP proxies and Web caches.

• HTTPFS also stands to benefit from various caching, load-balancing and replication facilities that web gateways offer.

• HTTPFS can rely on authentication mechanisms already built into Web servers, in addition to its own access control.

• HTTPFS can serve "files" and list "directories" that are created on the fly. In particular, HTTPFS permits browsing of a remote *database* as if it were a local filesystem.

• Whenever a remote file or directory get accessed or modified, HTTPFS can synchronously fire up triggers and run hooks. This is very difficult to accomplish with FTP.

See [4] for a description of another data-distribution service that builds upon HTTP riches. Design of a Linux-specific HTTP-based filesystem, in the context of WebDAV, userfs and perlfs, is discussed in [5].

Availability and installation

The MCFS/HTTPFS adapter distribution is freely available from a HTTPFS web page

http://pobox.com/~oleg/ftp/HTTP-VFS.html

The distribution archive contains the complete and self-contained source code for the server and the adapter, and an `INSTALL` document. A manifest file tells what all the other files are for.

I have personally run the MCHFS on HP-UX and SunSparc/Solaris with Netscape and Apache HTTP servers, and on Windows NT running IIS. The HTTPFS client – the MC/HTTPFS adapter in particular – ran on Sun/Solaris, HP-UX, and Linux platforms. The adapter successfully communicated with a Midnight Commander on a Linux host (MC version 4.1.36, as found in S.u.S.E. Linux distribution, versions 5 and 6).

I have not yet implemented the `unlink()`, `rename()`, `mkdir()`, and `chmod()` file system calls. I should also look into persistent HTTP connections and an option of transmitting only selected pieces of a requested file, which HTTP 1.1 allows (and encourages).

Summary: the OS *is* the browser

This article presents a poor-man's network file system, which is simple, very portable, and requires the least privileges to set up and run. HTTPFS offers a glimpse of one of Plan9's jewels – a uniform file-centric naming of disparate resources – but without Plan9. This file system showcases HTTP, which is capable of far more than merely carrying web pages. HTTP can aspire to be the kingpin protocol that glues computing, storage, etc. resources together to form a distributed system – the role 9P plays in Plan9 [6].

The design of HTTPFS suggests that, contrary to a cliche, it is the OS that is the browser. While Active Desktop lets you view local files and directories as if they were web pages, HTTPFS allows access to remote web pages and other resources as if they were local files. HTTPFS has all the attributes of an OS component: it implements (a broad subset of) the filesystem API; it maintains "vnodes" and "buffer caches"; it interacts with a persistent store and offers a uniform file-centric view of various remote resources. On the other hand, HTTPFS provides a superset of remote access services every Web browser has to implement on its own. The HTTPFS and other local and network filesystems manage storage and distribution of content, while an HTML formatter along with `xv`, `ghostscript` and similar applications provide interpretation and rendering of particular kinds of data. Thus as far as the OS is concerned, viewing a web page is to be thought similar to displaying an image file off an NFS-mounted disk, and searching the Web is no different than executing `find`/`grep` on a local filesystem.

References

[1] "HTTP Version 1.1," R. Fielding, J. Gettys, J. Mogul, H. Frystyk Nielsen, and T. Berners-Lee, January 1997. RFC-2068

[2] "Patch-free User-level Link-time intercepting of system calls and interposing on library functions," Oleg Kiselyov <http://pobox.com/~oleg/ftp/syscall-interpose.html>

[3] “The Midnight Commander” <http://www.gnome.org/mc/>

[4] "Pushing Weather Products via an HTTP pipe. Introduction to Metcast," Oleg Kiselyov <http://zowie.metnet.navy.mil/~spawar/JMV-TNG/>

[5] "An HTTP filesystem for Linux?" <http://rufus.w3.org/linux/httpfs/>

[6] "Plan 9 from Bell Labs," Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, Phil Winterbottom <http://plan9.bell-labs.com/plan9/doc/9.html>

Acknowledgement

Comments, suggestions, and shepherding by Chris Small are greatly appreciated.

Appendix A

Mapping between file system API and HTTP requests and responses

File System API call	**HTTP request issued**

`open filename-URL oflags mode`	`GET filename-URL`
	`Pragma: httpfs="preopen-xxxx"`
	`If-modified-since: yyyyy`

where `xxxx` encodes the file status flags and file access modes as given by oflags: `O_RDONLY`, `O_RDWR`, `O_WRONLY`, `O_CREAT`, `O_EXCL` and `O_TRUNC`. The HTTPFS server delivers the file if needed, and verifies that the resource can indeed be retrieved, modified, created or truncated. A `VNodeFile` is created to describe the opened resource and point to a local file that holds the (cached) copy of the resource. This local file is then opened, and the corresponding handle is returned to the caller.
If the file is being opened for modification, a `dirty` bit of the `VNodeFile` is set.
A `VNodeFile` corresponding to the `filename-URL` might have already existed in a `VNode` cache. In that case, the `GET` request will include an `If-modified-since: yyyyy` header, where `yyyyy` is the value of a `VNode::last_checked` field in HTTP date format.

`close cached-file-handle`	`PUT filename-URL`

Locate a `VNode` whose opened cache file has a handle equal to the `cached-file-handle`.
If the `filename-URL` has been opened for writing (that is, `VNodeFile::dirty` is set), upload the contents of the cache file to the HTTPFS server. The `VNode` and its cached content are not immediately disposed of, but rather stay around until "garbage-collected".

`read cached-file-handle buffer count`	None

Perform a regular `read(2)` operation on the `cached-file-handle.`

`write cached-file-handle buffer count`	None

Perform a regular `write(2)` operation on the `cached-file-handle`.

`lseek cached-file-handle offset whence`	None

Perform a regular `lseek(2)` operation on the `cached-file-handle`.

`stat filename-URL struct-stat-buffer`	`HEAD filename-URL` `Pragma: httpfs="stat"`

First we check to see if there is a valid `VNode` for the given `filename-URL` (possibly with a '`/`' appended, in case it turns out to be a directory). If such a `VNode` is found, its cached status information is immediately returned and a HTTPFS server is not bothered. Otherwise, we issue the `HEAD` request and fill in the `struct-stat-buffer` from the status-info* in a `Etag:` response header.

`lstat filename-URL struct-stat-buffer`	`HEAD filename-URL` `Pragma: httpfs="lstat"`

Similar to the `stat` API call above.

`readlink filename-URL filename-buffer`	`HEAD file-name` `Pragma: httpfs="readlink"`

Fill in the `filename-buffer` with the response from the server.

`opendir dirname-URL`	`GET dirname-URL` `If-modified-since: yyyyy`

A new `VNodeDir` is created for the `dirname-URL`, unless the corresponding valid `VNodeDir` happens to exist in the `VNode` cache. In the latter case, the `GET` request will carry the `If-modified-since: yyyyy` header with `yyyyy` being the value of a `VNode::last_checked` field.
The server returns the listing of the directory: for each directory entry (including . and ..) the server writes a line
`name/status-info`*
This listing is written as it is into a cache file of the `VNodeDir`. The `VHandle` of this `VNodeDir` is returned as the result of the `opendir()` call.

`readdir VNodeDir-handle`	None

The `dir-handle` is supposed to be a `VHandle` of a `VNodeDir`. This `VNode` is located, its cache file is parsed and sent to a MCFS client (as a sequence of `name, stat-for-the-name` pairs).

`closedir VNodeDir-handle`	None

The `VNodeDir-handle` is supposed to be a `VHandle` of a `VNodeDir`, which is thus closed.

`rmdir dirname-URL`	`DELETE dirname-URL`
`mkdir dirname-URL mode`	`PUT dirname-URL`
`unlink filename-URL`	`DELETE filename-URL`

* `status-info`, the status information for a remote resource, is a string of 11 numbers separated by a single space: "`dev ino mode nlink uid gid size atime mtime ctime blocks`". All numbers are in decimal notation, except `mode` which is octal. The meaning of the numbers is the same as that of the corresponding fields in a `stat` structure. See also a `stat` entry in Perl documentation. The `status-info` is a "hard validator" of a resource – resource's unique identification. Indeed, should the file be altered, at least its modification timestamp will change. The `status-info` is delivered in a `ETag:` response header, a field designated by the HTTP standard to carry (unique) resource identifiers.