NAME

fetchMakeURL, **fetchParseURL**, **fetchFreeURL**, **fetchXGetURL**, **fetchGetURL**, **fetchPutURL**, **fetchStatURL**, **fetchListURL**, **fetchXGet**, **fetchGet**, **fetchPut**, **fetchStat**, **fetchList**, **fetchXGetFile**, **fetchGetFile**, **fetchPutFile**, **fetchStatFile**, **fetchListFile**, **fetchXGetHTTP**, **fetchGetHTTP**, **fetchPutHTTP**, **fetchStatHTTP**, **fetchListHTTP**, **fetchReqHTTP**, **fetchXGetFTP**, **fetchGetFTP**, **fetchPutFTP**, **fetchStatFTP**, **fetchListFTP** - file transfer functions

LIBRARY

File Transfer Library (libfetch, -lfetch)

SYNOPSIS

#include <sys/param.h>
#include <stdio.h>
#include <fetch.h>

*struct url **
fetchMakeURL(*const char *scheme, const char *host, int port, const char *doc, const char *user, const char *pwd*);

*struct url **
fetchParseURL(*const char *URL*);

void
fetchFreeURL(*struct url *u*);

*FILE **
fetchXGetURL(*const char *URL, struct url_stat *us, const char *flags*);

*FILE **
fetchGetURL(*const char *URL, const char *flags*);

*FILE **
fetchPutURL(*const char *URL, const char *flags*);

int
fetchStatURL(*const char *URL, struct url_stat *us, const char *flags*);

*struct url_ent **
fetchListURL(*const char *URL, const char *flags*);

*FILE **
fetchXGet(*struct url *u, struct url_stat *us, const char *flags*);

*FILE **
fetchGet(*struct url *u, const char *flags*);

*FILE **
fetchPut(*struct url *u, const char *flags*);

int
fetchStat(*struct url *u, struct url_stat *us, const char *flags*);

*struct url_ent **
fetchList(*struct url *u, const char *flags*);

*FILE **
fetchXGetFile(*struct url *u, struct url_stat *us, const char *flags*);

*FILE **
fetchGetFile(*struct url *u, const char *flags*);

*FILE **
fetchPutFile(*struct url *u, const char *flags*);

int
fetchStatFile(*struct url *u, struct url_stat *us, const char *flags*);

*struct url_ent **
fetchListFile(*struct url *u, const char *flags*);

*FILE **
fetchXGetHTTP(*struct url *u, struct url_stat *us, const char *flags*);

*FILE **
fetchGetHTTP(*struct url *u, const char *flags*);

*FILE **
fetchPutHTTP(*struct url *u, const char *flags*);

int

fetchStatHTTP(*struct url *u, struct url_stat *us, const char *flags*);

*struct url_ent **
fetchListHTTP(*struct url *u, const char *flags*);

*FILE **
fetchReqHTTP(*struct url *u, const char *method, const char *flags, const char *content_type, const char *body*);

*FILE **
fetchXGetFTP(*struct url *u, struct url_stat *us, const char *flags*);

*FILE **
fetchGetFTP(*struct url *u, const char *flags*);

*FILE **
fetchPutFTP(*struct url *u, const char *flags*);

int
fetchStatFTP(*struct url *u, struct url_stat *us, const char *flags*);

*struct url_ent **
fetchListFTP(*struct url *u, const char *flags*);

DESCRIPTION

These functions implement a high-level library for retrieving and uploading files using Uniform Resource Locators (URLs).

fetchParseURL() takes a URL in the form of a null-terminated string and splits it into its components function according to the Common Internet Scheme Syntax detailed in RFC1738. A regular expression which produces this syntax is:

<scheme>:(//(<user>(:<pwd>)?@)?<host>(:<port>)?)?/(<document>)?

If the URL does not seem to begin with a scheme name, the following syntax is assumed:

((<user>(:<pwd>)?@)?<host>(:<port>)?)?/(<document>)?

Note that some components of the URL are not necessarily relevant to all URL schemes. For instance, the file scheme only needs the <scheme> and <document> components.

fetchMakeURL() and **fetchParseURL**() return a pointer to a *url* structure, which is defined as follows in <*fetch.h*>:

```
#define URL_SCHEMELEN 16
#define URL_USERLEN 256
#define URL_PWDLEN 256

struct url {
    char      scheme[URL_SCHEMELEN+1];
    char      user[URL_USERLEN+1];
    char      pwd[URL_PWDLEN+1];
    char      host[MAXHOSTNAMELEN+1];
    int               port;
    char     *doc;
    off_t     offset;
    size_t    length;
    time_t    ims_time;
};
```

The *ims_time* field stores the time value for If-Modified-Since HTTP requests.

The pointer returned by **fetchMakeURL**() or **fetchParseURL**() should be freed using **fetchFreeURL**().

fetchXGetURL(), **fetchGetURL**(), and **fetchPutURL**() constitute the recommended interface to the **fetch** library. They examine the URL passed to them to determine the transfer method, and call the appropriate lower-level functions to perform the actual transfer. **fetchXGetURL**() also returns the remote document's metadata in the *url_stat* structure pointed to by the *us* argument.

The *flags* argument is a string of characters which specify transfer options. The meaning of the individual flags is scheme-dependent, and is detailed in the appropriate section below.

fetchStatURL() attempts to obtain the requested document's metadata and fill in the structure pointed to by its second argument. The *url_stat* structure is defined as follows in <*fetch.h*>:

```
struct url_stat {
    off_t     size;
    time_t    atime;
    time_t    mtime;
};
```

If the size could not be obtained from the server, the *size* field is set to -1. If the modification time could not be obtained from the server, the *mtime* field is set to the epoch. If the access time could not be obtained from the server, the *atime* field is set to the modification time.

fetchListURL() attempts to list the contents of the directory pointed to by the URL provided. If successful, it returns a malloced array of *url_ent* structures. The *url_ent* structure is defined as follows in <*fetch.h*>:

```
struct url_ent {
    char        name[PATH_MAX];
    struct url_stat stat;
};
```

The list is terminated by an entry with an empty name.

The pointer returned by **fetchListURL**() should be freed using **free**().

fetchXGet(), **fetchGet**(), **fetchPut**() and **fetchStat**() are similar to **fetchXGetURL**(), **fetchGetURL**(), **fetchPutURL**() and **fetchStatURL**(), except that they expect a pre-parsed URL in the form of a pointer to a *struct url* rather than a string.

All of the **fetchXGetXXX**(), **fetchGetXXX**() and **fetchPutXXX**() functions return a pointer to a stream which can be used to read or write data from or to the requested document, respectively. Note that although the implementation details of the individual access methods vary, it can generally be assumed that a stream returned by one of the **fetchXGetXXX**() or **fetchGetXXX**() functions is read-only, and that a stream returned by one of the **fetchPutXXX**() functions is write-only.

FILE SCHEME

fetchXGetFile(), **fetchGetFile**() and **fetchPutFile**() provide access to documents which are files in a locally mounted file system. Only the <document> component of the URL is used.

fetchXGetFile() and **fetchGetFile**() do not accept any flags.

fetchPutFile() accepts the ‘a’ (append to file) flag. If that flag is specified, the data written to the stream returned by **fetchPutFile**() will be appended to the previous contents of the file, instead of replacing them.

FTP SCHEME

fetchXGetFTP(), **fetchGetFTP**() and **fetchPutFTP**() implement the FTP protocol as described in RFC959.

If the ‘P’ (not passive) flag is specified, an active (rather than passive) connection will be attempted.

The ‘p’ flag is supported for compatibility with earlier versions where active connections were the default. It has precedence over the ‘P’ flag, so if both are specified, **fetchMakeURL** will use a passive connection.

If the ‘l’ (low) flag is specified, data sockets will be allocated in the low (or default) port range instead of the high port range (see ip(4)).

If the ‘d’ (direct) flag is specified, **fetchXGetFTP**(), **fetchGetFTP**() and **fetchPutFTP**() will use a direct connection even if a proxy server is defined.

If no user name or password is given, the **fetch** library will attempt an anonymous login, with user name "anonymous" and password "anonymous@<hostname>".

HTTP SCHEME

The **fetchXGetHTTP**(), **fetchGetHTTP**(), **fetchPutHTTP**() and **fetchReqHTTP**() functions implement the HTTP/1.1 protocol. With a little luck, there is even a chance that they comply with RFC2616 and RFC2617.

If the ‘d’ (direct) flag is specified, **fetchXGetHTTP**(), **fetchGetHTTP**() and **fetchPutHTTP**() will use a direct connection even if a proxy server is defined.

If the ‘i’ (if-modified-since) flag is specified, and the *ims_time* field is set in *struct url*, then **fetchXGetHTTP**() and **fetchGetHTTP**() will send a conditional If-Modified-Since HTTP header to only fetch the content if it is newer than *ims_time*.

The function **fetchReqHTTP**() can be used to make requests with an arbitrary HTTP verb, including POST, DELETE, CONNECT, OPTIONS, TRACE or PATCH. This can be done by setting the argument *method* to the intended verb, such as ‘POST’, and *body* to the content.

Since there seems to be no good way of implementing the HTTP PUT method in a manner consistent with the rest of the **fetch** library, **fetchPutHTTP**() is currently unimplemented.

HTTPS SCHEME

Based on HTTP SCHEME. By default the peer is verified using the CA bundle located in */usr/local/etc/ssl/cert.pem*. If this file does not exist, */etc/ssl/cert.pem* is used instead. If neither file exists, and SSL_CA_CERT_PATH has not been set, OpenSSL’s default CA cert and path settings apply. The certificate bundle can contain multiple CA certificates. A common source of a current CA bundle is *security/ca_root_nss*.

The CA bundle used for peer verification can be changed by setting the environment variables SSL_CA_CERT_FILE to point to a concatenated bundle of trusted certificates and SSL_CA_CERT_PATH to point to a directory containing hashes of trusted CAs (see verify(1)).

A certificate revocation list (CRL) can be used by setting the environment variable SSL_CRL_FILE (see crl(1)).

Peer verification can be disabled by setting the environment variable SSL_NO_VERIFY_PEER. Note that this also disables CRL checking.

By default the service identity is verified according to the rules detailed in RFC6125 (also known as hostname verification). This feature can be disabled by setting the environment variable SSL_NO_VERIFY_HOSTNAME.

Client certificate based authentication is supported. The environment variable SSL_CLIENT_CERT_FILE should be set to point to a file containing key and client certificate to be used in PEM format. When a PEM-format key is in a separate file from the client certificate, the environment variable SSL_CLIENT_KEY_FILE can be set to point to the key file. In case the key uses a password, the user will be prompted on standard input.

By default **libfetch** allows TLSv1 and newer when negotiating the connecting with the remote peer. You can change this behavior by setting the SSL_NO_TLS1, SSL_NO_TLS1_1 and SSL_NO_TLS1_2 environment variables to disable TLS 1.0, 1.1 and 1.2 respectively.

AUTHENTICATION

Apart from setting the appropriate environment variables and specifying the user name and password in the URL or the *struct url*, the calling program has the option of defining an authentication function with the following prototype:

int **myAuthMethod**(*struct url *u*)

The callback function should fill in the *user* and *pwd* fields in the provided *struct url* and return 0 on success, or any other value to indicate failure.

To register the authentication callback, simply set *fetchAuthMethod* to point at it. The callback will be used whenever a site requires authentication and the appropriate environment variables are not set.

This interface is experimental and may be subject to change.

RETURN VALUES

fetchParseURL() returns a pointer to a *struct url* containing the individual components of the URL. If it is unable to allocate memory, or the URL is syntactically incorrect, **fetchParseURL**() returns a NULL pointer.

The **fetchStat**() functions return 0 on success and -1 on failure.

All other functions return a stream pointer which may be used to access the requested document, or NULL if an error occurred.

The following error codes are defined in <*fetch.h*>:

[FETCH_ABORT] Operation aborted

[FETCH_AUTH] Authentication failed

[FETCH_DOWN] Service unavailable

[FETCH_EXISTS] File exists

[FETCH_FULL] File system full

[FETCH_INFO] Informational response

[FETCH_MEMORY] Insufficient memory

[FETCH_MOVED] File has moved

[FETCH_NETWORK]
Network error

[FETCH_OK] No error

[FETCH_PROTO] Protocol error

[FETCH_RESOLV] Resolver error

[FETCH_SERVER] Server error

[FETCH_TEMP] Temporary error

[FETCH_TIMEOUT] Operation timed out

[FETCH_UNAVAIL] File is not available

[FETCH_UNKNOWN]
Unknown error

[FETCH_URL] Invalid URL

The accompanying error message includes a protocol-specific error code and message, like "File is not available (404 Not Found)"

ENVIRONMENT

FETCH_BIND_ADDRESS Specifies a hostname or IP address to which sockets used for outgoing connections will be bound.

FTP_LOGIN Default FTP login if none was provided in the URL.

FTP_PASSIVE_MODE If set to ‘no’, forces the FTP code to use active mode. If set to any other value, forces passive mode even if the application requested active mode.

FTP_PASSWORD Default FTP password if the remote server requests one and none was provided in the URL.

FTP_PROXY URL of the proxy to use for FTP requests. The document part is ignored. FTP and HTTP proxies are supported; if no scheme is specified, FTP is assumed. If the proxy is an FTP proxy, **libfetch** will send ‘user@host’ as user name to the proxy, where ‘user’ is the real user name, and ‘host’ is the name of the FTP server.

If this variable is set to an empty string, no proxy will be used for FTP requests, even if the HTTP_PROXY variable is set.

ftp_proxy Same as FTP_PROXY, for compatibility.

HTTP_ACCEPT Specifies the value of the *Accept* header for HTTP requests. If empty, no *Accept* header is sent. The default is "*/*".

HTTP_AUTH Specifies HTTP authorization parameters as a colon-separated list of items. The first and second item are the authorization scheme and realm

respectively; further items are scheme-dependent. Currently, the "basic" and "digest" authorization methods are supported.

Both methods require two parameters: the user name and password, in that order.

This variable is only used if the server requires authorization and no user name or password was specified in the URL.

HTTP_PROXY — URL of the proxy to use for HTTP requests. The document part is ignored. Only HTTP proxies are supported for HTTP requests. If no port number is specified, the default is 3128.

Note that this proxy will also be used for FTP documents, unless the FTP_PROXY variable is set.

http_proxy — Same as HTTP_PROXY, for compatibility.

HTTP_PROXY_AUTH — Specifies authorization parameters for the HTTP proxy in the same format as the HTTP_AUTH variable.

This variable is used if and only if connected to an HTTP proxy, and is ignored if a user and/or a password were specified in the proxy URL.

HTTP_REFERER — Specifies the referrer URL to use for HTTP requests. If set to "auto", the document URL will be used as referrer URL.

HTTP_USER_AGENT — Specifies the User-Agent string to use for HTTP requests. This can be useful when working with HTTP origin or proxy servers that differentiate between user agents. If defined but empty, no User-Agent header is sent.

NETRC — Specifies a file to use instead of *~/.netrc* to look up login names and passwords for FTP and HTTP sites as well as HTTP proxies. See ftp(1) for a description of the file format.

NO_PROXY — Either a single asterisk, which disables the use of proxies altogether, or a comma- or whitespace-separated list of hosts for which proxies should not be used.

no_proxy — Same as NO_PROXY, for compatibility.

SOCKS5_PROXY Uses SOCKS version 5 to make connection. The format must be the IP or hostname followed by a colon for the port. IPv6 addresses must enclose the address in brackets. If no port is specified, the default is 1080. This setting will supercede a connection to an HTTP_PROXY.

SSL_CA_CERT_FILE CA certificate bundle containing trusted CA certificates. Default value: See HTTPS SCHEME above.

SSL_CA_CERT_PATH Path containing trusted CA hashes.

SSL_CLIENT_CERT_FILE
PEM encoded client certificate/key which will be used in client certificate authentication.

SSL_CLIENT_KEY_FILE PEM encoded client key in case key and client certificate are stored separately.

SSL_CRL_FILE File containing certificate revocation list.

SSL_NO_TLS1 Do not allow TLS version 1.0 when negotiating the connection.

SSL_NO_TLS1_1 Do not allow TLS version 1.1 when negotiating the connection.

SSL_NO_TLS1_2 Do not allow TLS version 1.2 when negotiating the connection.

SSL_NO_VERIFY_HOSTNAME
If set, do not verify that the hostname matches the subject of the certificate presented by the server.

SSL_NO_VERIFY_PEER If set, do not verify the peer certificate against trusted CAs.

EXAMPLES

To access a proxy server on *proxy.example.com* port 8080, set the HTTP_PROXY environment variable in a manner similar to this:

```
HTTP_PROXY=http://proxy.example.com:8080
```

If the proxy server requires authentication, there are two options available for passing the authentication data. The first method is by using the proxy URL:

HTTP_PROXY=http://<user>:<pwd>@proxy.example.com:8080

The second method is by using the HTTP_PROXY_AUTH environment variable:

HTTP_PROXY=http://proxy.example.com:8080
HTTP_PROXY_AUTH=basic:*:<user>:<pwd>

To disable the use of a proxy for an HTTP server running on the local host, define NO_PROXY as follows:

NO_PROXY=localhost,127.0.0.1

To use a SOCKS5 proxy, set the SOCKS5_PROXY environment variable to a valid host or IP followed by an optional colon and the port. IPv6 addresses must be enclosed in brackets. The following are examples of valid settings:

SOCKS5_PROXY=proxy.example.com
SOCKS5_PROXY=proxy.example.com:1080
SOCKS5_PROXY=192.0.2.0
SOCKS5_PROXY=198.51.100.0:1080
SOCKS5_PROXY=[2001:db8::1]
SOCKS5_PROXY=[2001:db8::2]:1080

Access HTTPS website without any certificate verification whatsoever:

SSL_NO_VERIFY_PEER=1
SSL_NO_VERIFY_HOSTNAME=1

Access HTTPS website using client certificate based authentication and a private CA:

SSL_CLIENT_CERT_FILE=/path/to/client.pem
SSL_CA_CERT_FILE=/path/to/myca.pem

SEE ALSO

fetch(1), ip(4)

J. Postel and J. K. Reynolds, *File Transfer Protocol*, October 1985, RFC959.

P. Deutsch, A. Emtage, and A. Marine., *How to Use Anonymous FTP*, May 1994, RFC1635.

T. Berners-Lee, L. Masinter, and M. McCahill, *Uniform Resource Locators (URL)*, December 1994, RFC1738.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol -- HTTP/1.1*, January 1999, RFC2616.

J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, *HTTP Authentication: Basic and Digest Access Authentication*, June 1999, RFC2617.

HISTORY

The **fetch** library first appeared in FreeBSD 3.0.

AUTHORS

The **fetch** library was mostly written by Dag-Erling Smørgrav <*des@FreeBSD.org*> with numerous suggestions and contributions from Jordan K. Hubbard <*jkh@FreeBSD.org*>, Eugene Skepner <*eu@qub.com*>, Hajimu Umemoto <*ume@FreeBSD.org*>, Henry Whincup <*henry@techiebod.com*>, Jukka A. Ukkonen <*jau@iki.fi*>, Jean-François Dockes <*jf@dockes.org*>, Michael Gmelin <*freebsd@grem.de*> and others. It replaces the older **ftpio** library written by Poul-Henning Kamp <*phk@FreeBSD.org*> and Jordan K. Hubbard <*jkh@FreeBSD.org*>.

This manual page was written by Dag-Erling Smørgrav <*des@FreeBSD.org*> and Michael Gmelin <*freebsd@grem.de*>.

BUGS

Some parts of the library are not yet implemented. The most notable examples of this are **fetchPutHTTP**(), **fetchListHTTP**(), **fetchListFTP**() and FTP proxy support.

There is no way to select a proxy at run-time other than setting the HTTP_PROXY or FTP_PROXY environment variables as appropriate.

libfetch does not understand or obey 305 (Use Proxy) replies.

Error numbers are unique only within a certain context; the error codes used for FTP and HTTP overlap, as do those used for resolver and system errors. For instance, error code 202 means "Command not implemented, superfluous at this site" in an FTP context and "Accepted" in an HTTP context.

fetchStatFTP() does not check that the result of an MDTM command is a valid date.

In case password protected keys are used for client certificate based authentication the user is prompted for the password on each and every fetch operation.

The man page is incomplete, poorly written and produces badly formatted text.

The error reporting mechanism is unsatisfactory.

Some parts of the code are not fully reentrant.