edtFTPj

FTP Client Java Component

Developer's Guide

edtFTPj - A component of
the EDT **Platform**

Enterprise Distributed Technologies
http://www.enterprisedt.com

Table of Contents

1 Introduction

The design philosophy behind the edtFTPj API is to closely mirror the standard FTP commands as defined by RFC959. The advantage of this approach is that most Java developers already know how to use FTP and therefore, by implication, already know the edtFTPj interface.

The purpose of this guide is to describe the usage of the edtFTPj API in embedding FTP functionality in Java applications.

Note that a secure version of edtFTPj, called edtFTPj/SSL, is available. This implements the edtFTPj API over SSL, a protocol known as FTPS. If you require secure FTP, please consider using edtFTPj/SSL. Note that edtFTPj/SSL is a commercial product, with source code optionally available.

2 FTP Protocol Overview

2.1 Introduction

FTP (File Transfer Protocol) is a well established Internet protocol designed to transfer files (and information about files) across networks using TCP (Transmission Control Protocol).

FTP is defined in the Request For Comments 959 document (RFC 959), which can be obtained from the Internet Engineering Task Force.

FTP requires a client program (FTP client) and a server program (FTP server). The client can fetch files and file details from the server, and also upload files to the server. The server is generally password protected.

FTP commands are initiated by the client, which opens a TCP connection called the control connection to the server. This control connection is used for the entire duration of a session between the client and server. A session typically begins when the client logs in, and ends when the quit command is sent to the server. The control connection is used exclusively for sending FTP commands and reading server replies - it is never used to transfer files.

Transient TCP connections called data connections are set up whenever data (normally a file's contents) is to be transferred. For example, the client issues a command to retrieve a file from the server via the control channel. A data connection is then established, and the file's contents transferred to the client across it. Once the transfer is complete, the data connection is closed. Meanwhile, the control connection is maintained.

2.2 Active and passive modes

Data connections may be set up in two different ways, active and passive. Note that active and passive refer to the operation of the FTP server, not the client.

2.2.1 Passive mode

In passive mode, the client sends a PASV command to the server. This tells the server to listen for a connection attempt from the client, hence the server is passively waiting. The server replies to PASV with the host and port address that the server is listening on. The client deciphers this reply and when a data connection is required, attempts to initiate the connection to the server at this address.

2.2.2 Active mode

In active mode, the server actively connects to the client. To set up active mode, the client sends a PORT command to the server, specify the address and port number the client is listening on. When a data connection is required, the server initiates a connection to the client at this address.

Generally the server is responsible for closing data connections.

2.3 FTP Commands

FTP commands sent across the control connection consist of simple text strings (and follow the Telnet protocol - see RFC 854). For example, to retrieve a file, the client sends "RETR filename" on the control connection to the FTP server. To transfer a file, the client sends "STOR filename".

The FTP server acknowledges each command with an FTP reply, which consists of a three digit number followed by human-readable text. The first digit indicates if the response is good, bad, or incomplete. If an error occurred, the second digit may be used to indicate what type of error occurred. Similarly, the third digit can indicate more details of the error.

The first digit is the most important, and the five possible values are described below:

Reply	***Description***
1yz	Positive Preliminary reply. The request action has been initiated, but another reply is to be expected before the client issues another command.
2yz	Positive Completion reply. The requested action has successfully completed, and the client may issue another command.
3yz	Positive Intermediate reply. The command has been accepted, but more information is required. The client should send another command in reply.
4yz	Transient Negative reply. The command failed, but it can be retried
5yz	Permanent Negative Completion reply. The command failed, and should not be repeated.

2.4 Sample scenarios

2.4.1 Example 1

For example, to change directory the client sends:

```
> CWD dirname
```

The server responds with:

```
250 CWD command successful
```

As the reply begins with a ‘2’, we know the command sequence is completed.

However if we attempt to change directory to one that does not exist:

```
> CWD nonexistentdir
```

The server responds with:

```
550 nonexistentdir: The system cannot find the file specified.
```

As the reply begins with a ‘5’ we know that the command failed, and that it will fail again if repeated (unless the missing directory is created on the server).

2.4.2 Example 2

To transfer a text file, we issue a ‘RETR’ command to the server. However to transfer the file we require a data connection to be set up. This can be done using active or passive mode.

As discussed previously, in active mode, the client sends a ‘PORT’ command, indicating what address and port number the server should connect to:

```
> PORT 192,168,10,1,4,93
```

The first four digits are the IP address, and the last two encode the port number (in two 8-bit fields, the first being the high order bits of the 16-bit port number).

The server responds with:

```
200 PORT command successful.
```

This indicates that the data connection has been established.

Next, the ‘RETR’ command is issued:

```
> RETR abc.txt
```

The server responds with:

```
150 Opening ASCII mode data connection for abc.txt(70776 bytes).
```

The reply begins with a ‘1’, so we know that the command was successful, but the server will be sending another reply – the client cannot issue another command until this is received.

Eventually, the server sends:

```
226 Transfer complete.
```

The command sequence is complete, the file has been transferred, and the client can issue another command.

See RFC 959 for details about the second digit, and more extensive examples.

Note that most standard command-line FTP clients support debug mode, which displays the FTP commands that are being sent to the server, and the reply strings that are received back. Typing “debug” at the prompt will usually put the client into debug mode.

2.5 Data types

The two most common data types in usage are ASCII and binary.

ASCII is the default data type, and is intended for the transfer of text files. A line of text is followed by <CRLF>. Note that different operating systems use different end of line terminators.

Binary type (known as IMAGE in FTP) is used to transfer binary files. A byte-by-byte copy is made of the source file. Graphical images, video and executable files are all binary files. If they are transferred as ASCII, they will be corrupted.

2.6 Session commands

To begin an FTP session, the USER command is sent to the server:

```
> USER javaftp
```

The server responds with:

```
331 Password required for javaftp.
```

The client must respond with the password:

```
> PASS mypassword
```

The server responds with:

```
230 User javaftp logged in.
```

The session is now established, and the user can begin issuing various file and directory-related commands.

To end the session, the client sends:

```
> QUIT
```

The server responds with:

```
221
```

The session is now closed, and any further attempt to send commands to the server will fail.

2.7 File commands

FTP supports numerous file-related commands.

Files can be deleted (DELE) and renamed (RNFR,RNTO) as well as stored (STOR) and retrieved (RETR). When a file is stored, it can be written over or appended to (APPE).

See the Sample scenarios examples for more details.

2.8 Directory commands

FTP supports a variety of directory-related commands.

Directories can be created (MKD), removed (RMD), or changed into (CWD, CDUP).

Two types of directory listings can be made with FTP.

The LIST method obtains a list of files (and possibly directories). If a directory is specified, the server returns a list of files in the directory, together with system specific information about the files. The file information sent will vary depending on the server system. The data type should be set to ASCII for this file name list. If no directory is specified, details of the current working directory listing are sent.

The NAME LIST (NLST) method is similar to LIST, but only file names are returned. No other information about the files is sent. Again, the data type should be set to ASCII.

3 edtFTPj API

This section will provide an overview of the major features of the library directly related to FTP, which are encapsulated in the edtFTPj API. Developers are referred to edtFTPj's JavaDoc for a detailed API description.

The edtFTPj API is a relatively thin veneer over the FTP protocol described earlier.

Almost all methods can throw an `IOException` or an `FTPException`. The `FTPException` provides a method for obtaining the reply code (`getReplyCode()`).

3.1 Constructors

`FTPClient` is the main interface to the edtFTPj library, and in many cases, it is the only class which the developer needs to use.

All constructors other than the default constructor are now deprecated, and setter methods should be used to set the remote host and other parameters such as timeout (for socket reads and writes, specified in milliseconds), and the control port, which allows a different control port to the standard FTP port 21 to be specified.

3.2 Login

The `login()` method permits logging in to a remote FTP account, supplying a user name and password. The `user()` and `password()` methods perform the equivalent operations, and are supplied separately in the event that a server might be configured to require no password.

The `quit()` method quits the current FTP session.

3.3 Directory navigation and control

A number of methods are provided for remote directory navigation. Locally, the current directory of the calling application is used, and no methods are provided for local directory navigation.

The `chdir()` method changes the current remote directory to the one specified, while pwd() returns the current directory.

The `mkdir()` method creates a new directory, while the `rmdir()` method deletes a directory (which for most FTP servers must be empty).

Directories (and files) can be renamed using the `rename()` method.

3.4 Files and file transfers

The edtFTPj supports a varied API for file operations.

Files can be deleted with `delete()`, and renamed via `rename()`.

If supported by the FTP server, the `size()` command can be used to determine the size (in bytes) of a remote file. Similarly, the `modtime()` can determine the modification time of a remote file, if supported. The returned `java.lang.Date` is in GMT.

The various `put()` and `get()` methods are supplied to put files onto the remote server, and to retrieve them.

In their simplest form, the local and remote file names are supplied as parameters, and the file contents transferred. If putting, files are placed in the remote current directory. If getting, files are fetched from the remote current directory.

The transfer mode is important – both ASCII and binary modes are supported as discussed in the

FTP protocol section. Binary mode is simply a byte by byte exact copy of the file to be transferred. Typically, this would be used for binary files such as images, executables and documents in binary form (e.g. Microsoft Word or Adobe Acrobat files).

ASCII mode attempts to translate end of line characters in text files, which vary between operating systems. For example, Windows uses a `CRLF` to indicate the end of a line, while Unix uses just LF. Consequently, ASCII transfers can result in the local and remote files being different sizes. Also, transferring a binary file in ASCII mode will corrupt the file (by inserting spurious `CR`s, for example).

The transfer modes are encapsulated in the `FTPTransferType` class, which has ASCII and BINARY as final static variables.

The initial transfer type is always ASCII, and the current transfer type can be found using `getType()`. Similarly, `setType()` sets the current transfer type.

Other `get()` and `put()` methods are also supplied. One form permits an `OutputStream` (for `get()`s), or an `InputStream` (for `put()`s) to be supplied in lieu of a local file name.

Bytes buffers can be supplied as destinations (for `get()`s) and sources (for `put()`s) – convenient if you don't wish to write a file out to disk.

All `put()` commands also offer a variation that can direct that the contents being transferred are to be appended to any existing remote file. Normally, remote files are overwritten. Not all FTP servers support or permit append operations – they may need to be configured for this option. Obviously, there are potential security risks associated with appending, as a rogue client could continually do so until disk space on the server was exhausted.

All transfer commands can be canceled during their operation by the `cancelTransfer()` method.

Also, the `setProgressMonitor()` method can be used to set up a progress monitor that reports regularly during transfers. Typically, the `FTPProgressMonitor` interface is implemented, and the `bytesTransferred()` method is called every `n` bytes that are transferred.

3.5 Directory listings

Obtaining remote directory listings is an important part of the API.

The `dir()` method is used to retrieve simple listings as an array of strings. If no directory name is supplied, the current directory is listed. A directory name (that is visible from the remote current directory) can also be supplied.

If the '`full`' parameter is used and set to `true`, a detailed listing is returned. Each string contains a variety of server-dependent details about the file or directory.

The `dirDetails()` method works in the same way as when a full listing is requested via `dir()`, but attempts to interpret the detailed listing and construct an `FTPFile` object for each remote file or directory. The `FTPFile` object indicates the file name, size, whether or not it is a directory and other useful attributes.

Some FTP servers permit file masks to be used in dir(), however many do not. It is best not to assume this is possible. Similarly, some FTP servers do not even permit a directory to be supplied, but simply list the current directory.

3.6 Miscellaneous commands

There are a number of useful additional commands that fall into no particular category.

The `system()` command returns a string representing the remote operating system. This is server dependent.

The `help()` command returns the remote server's help text for the supplied command.

The current version of the library can be obtained via `getVersion()`, which returns an array of three integers, representing {major,middle,minor} version numbers. For example, version 1.4.0 of the library would return `{1,4,0}`. The `getBuildTimestamp()` method returns the timestamp of the library's build, in `d-MMM-yyyy HH:mm:ss z` format.

3.7 Message Logging

Sometimes it is useful to obtain a log of the FTP commands and replies that are sent back and forth. These are available via the logging API, but occasionally it is useful to get a separate record of messages that is not mixed in with the logging (which can be extensive depending on what level is set).

The `FTPMessageListener` is for precisely this purpose. It defines an interface which has methods for logging commands and their replies (`logCommand()` and `logReply()`). Developers can implement this interface and set the listener for an instance of `FTPClient` via `setMessageListener()`. Then all FTP messages can be collected, for example, in a `StringBuffer` or perhaps a `List`.

For convenience, `FTPMessageCollector` is an implementation of `FTPMessageListener` provided in the `com.enterprisedt.net.ftp` package. It simply collects messages in a `StringBuffer`, which can be obtained by calling `getLog()`.

3.8 Firewalls

Often, a firewall and/or proxy separates FTP clients from FTP servers. Sometimes, it is necessary to perform some configuration (perhaps even in the firewall) so that the server can be reached.

SOCKS proxy servers are relatively easy to connect through. The `initSOCKS()` method on the `FTPClient` class can be used to set the hostname and port of the SOCKS server. If username/password authentication is required, `initSOCKSAuthentication()` can be used to set the username and password for the SOCKS server. Note however that these static methods set the SOCKS properties for the entire Java Virtual Machine – it is not possible to set them for a specific FTPClient instance.

There is also a standard protocol for using a firewall/proxy server to connect to a remote host. When constructing an instance of FTPClient, instead of supplying the remote hostname, the name of the firewall is supplied instead. Instead of supplying the remote user as the user, a string is supplied in the form of remoteuser@remotehost.com. The remote password is supplied as normal.

So instead of supplying ftp.remotehost.com to the FTPClient constructor (amongst other parameters), firewallhostname is supplied. Similarly, instead of supplying (remoteuser, remotepassword) to the `login()` method, (remoteuser@remotehost.com, remotepassword) is supplied. Many firewalls support this syntax. Firewalls we have confirmed support this syntax at the time of writing include Check Point Firewall-1 and WinProxy.

4 Logging API

4.1 Overview

edtFTPj has a powerful logging API modelled on the popular log4j library – in fact full integration with log4j is supported.

The Java package is `com.enterprisedt.util.debug`, and the key class is the Logger class. See the JavaDoc for the full API.

The logging API allows developers to embed log statements in their code that can disabled or enabled at runtime, and can be directed to standard output, or to file or both. Stack traces from caught exceptions are an optional parameter to logging calls. Judicious use of logging in applications can be a powerful aid to debugging, as well as a useful tool for diagnosing production problems. Also, all log statements are prefixed with a timestamp accurate to milliseconds, and can thus be extremely valuable in performance tuning.

4.2 Logging Levels

A number of logging levels are supported - FATAL,ERROR,WARN,INFO,DEBUG (as well as ALL and OFF, which are not really levels as such). Logging statements are made at a certain level, such as INFO, and the level for the library as a whole is set. DEBUG is considered to be the highest level of logging (as it generally produces the most output). A message is logged if its level is less than or equal to the overall logging level. For example, if a logging call is made at the DEBUG level and the overall level is set to INFO, it would not appear in the logs. If a logging call is made at WARN, and the overall level is set to INFO, it would appear in the logs.

Logging levels are encapsulated in the Level class. For example, the WARN level is represented by Level.WARN.

By default, all logging output is directed to the standard output stream (if it is equal to or less than the level set for the library). Appenders are supported, whereby output can be directed elsewhere. Currently, FileAppenders are the only implemented Appenders, which permit logging output to be directed to named files.

4.3 Configuration

By default, the log level is switched to OFF, so that no logging will appear.

The log level can be changed in two ways. Firstly, it can be changed explicitly by calling the setLevel() method on the Logger class. For example,

```
Logger.setLevel(Level.DEBUG);
```

will set the global logging level to DEBUG.

A System property, **`edtftp.log.level`**, can also be used to set the logging level. For example, using the -D option to set an application's System property, you could use

```
java -Dedtftp.log.level=INFO com.mypackage.myclass
```

to set the global logging level to INFO.

Full integration with log4j is possible. A System property, **`edtftp.log.log4j`**, is used to indicate that log4j integration should be attempted. It must be set to “true”. Also, the log4j jar file must be available in the CLASSPATH. Once this is done all logging calls are directed via log4j, using reflection, and the standard log4j settings are used. More details on log4j can be found at the log4j site listed in the references.

System Property	Description	Values
`edtftp.log.level`	Controls the global logging level	FATAL,ERROR, WARN,INFO,DEBUG, ALL,OFF
`edtftp.log.log4j`	Controls log4j integration.	true or false

4.4 Examples

To add logging to a particular class, a Logger object is required. Typically, an instance of a Logger is created for every class that logs – a static class member, as below:

```
private static Logger log = Logger.getLogger(MyClass.class);
```

Logging can then be performed by calling the logging methods on the log object, e.g.

```
log.info("Connecting to server " + host);

log.debug("User name: " + user);
```

Note that exceptions can be passed to the logging methods as a second parameter, and the stack trace is written to the log stream, e.g.

```
try {

    ... do something ...

}

catch (Exception ex) {
    log.error("Failed to do something", ex);

}
```

Although the overall logging level can be set to, say INFO, so that log.debug() calls are not sent to the log stream, sometimes constructing calls can be expensive if they involve a number of string concatenations. In these cases, it is best to test the logging level before making the call, so that the expensive construction is never performed, e.g.

```
if (log.isDebugEnabled())

      log.debug(“Transferred “ + n + “ bytes for file “ + filename);
```

As noted, all logging by default goes to standard out. A FileAppender must be added if logging is to go to a file (and this will disable logging to standard out). An example is shown below:

```
Logger.addAppender(new FileAppender(myLogFileName));
```

Now all logging output will go to the FileAppender's file, and no logging will go to standard output. Multiple FileAppenders can be added. If the StandardOutputAppender is added to the Logger as well, logging will be directed to the file and to standard output.

5 Glossary

API – Application Programmer's Interface

Base-64 Encoding – An encoding use for transmitting binary data through a text only communication channel – only readable characters are used.

Control Channel – TCP connection used by an FTP session to transmit commands and the responses to those commands. There is only one of these per session.

CRLF – Carriage return/Line Feed.

Data Channel – TCP connection used by an FTP session to transmit data such as a directory listing or a file. There are multiple data channels per session; one for each data transfer.

EDT – Enterprise Distributed Technologies

IETF – Internet Engineering Task Force.

FTP – File Transfer Protocol

FTPS – FTP over SSL/TLS. It comes in 2 variants: explicit FTPS and implicit FTPS (see the definitions for those terms in this glossary).

Internet Draft – Working document of the IETF. An Internet Draft may eventually be approved as an RFC. Internet Drafts often become defacto standards before they are approved as RFCs due to the length of the process of RFC approval.

RFC – Request For Comment. A formal Internet specification approved by the IETF.

RFC959 – The specification for FTP.

SSL – Secure Sockets Layer.

TLS – Transport Layer Security (effectively the latest version of SSL).

6 References

Document	*Location*
IETF	http://www.ietf.org
RFC959	http://www.faqs.org/rfcs/rfc959.html
Log4j documentation	http://logging.apache.org/log4j/docs/