NAME

ffmpeg−protocols − FFmpeg protocols

DESCRIPTION

This document describes the input and output protocols provided by the libavformat library.

PROTOCOL OPTIONS

The libavformat library provides some generic global options, which can be set on all the protocols. In addition each protocol may support so-called private options, which are specific for that component.

Options may be set by specifying −*option value* in the FFmpeg tools, or by setting the value explicitly in the `AVFormatContext` options or using the *libavutil/opt.h* API for programmatic use.

The list of supported options follows:

protocol_whitelist *list* **(***input***)**

Set a ‘,’−separated list of allowed protocols. ‘ALL’ matches all protocols. Protocols prefixed by ‘−’ are disabled. All protocols are allowed by default but protocols used by an another protocol (nested protocols) are restricted to a per protocol subset.

PROTOCOLS

Protocols are configured elements in FFmpeg that enable access to resources that require specific protocols.

When you configure your FFmpeg build, all the supported protocols are enabled by default. You can list all available ones using the configure option ‘−−list−protocols’.

You can disable all the protocols using the configure option ‘−−disable−protocols’, and selectively enable a protocol using the option "−−enable−protocol=*PROTOCOL*‘, or you can disable a particular protocol using the option ’’−−disable−protocol=*PROTOCOL*".

The option ‘−protocols’ of the ff* tools will display the list of supported protocols.

All protocols accept the following options:

rw_timeout

Maximum time to wait for (network) read/write operations to complete, in microseconds.

A description of the currently available protocols follows.

amqp

Advanced Message Queueing Protocol (AMQP) version 0−9−1 is a broker based publish-subscribe communication protocol.

FFmpeg must be compiled with −−enable−librabbitmq to support AMQP. A separate AMQP broker must also be run. An example open-source AMQP broker is RabbitMQ.

After starting the broker, an FFmpeg client may stream data to the broker using the command:

```
ffmpeg −re −i input −f mpegts amqp://[[user]:[password]@]hostname[:port][
```

Where hostname and port (default is 5672) is the address of the broker. The client may also set a user/password for authentication. The default for both fields is ‘guest’. Name of virtual host on broker can be set with vhost. The default value is ‘/’.

Muliple subscribers may stream from the broker using the command:

```
ffplay amqp://[[user]:[password]@]hostname[:port][/vhost]
```

In RabbitMQ all data published to the broker flows through a specific exchange, and each subscribing client has an assigned queue/buffer. When a packet arrives at an exchange, it may be copied to a client’s queue depending on the exchange and routing_key fields.

The following options are supported:

exchange

Sets the exchange to use on the broker. RabbitMQ has several predefined exchanges: ‘amq.direct’ is the default exchange, where the publisher and subscriber must have a matching routing_key; ‘amq.fanout’ is the same as a broadcast operation (i.e. the data is forwarded to all queues on the

fanout exchange independent of the routing_key); and ‘‘amq.topic’’ is similar to ‘‘amq.direct’’, but allows for more complex pattern matching (refer to the RabbitMQ documentation).

routing_key
Sets the routing key. The default value is ‘‘amqp’’. The routing key is used on the ‘‘amq.direct’’ and ‘‘amq.topic’’ exchanges to decide whether packets are written to the queue of a subscriber.

pkt_size
Maximum size of each packet sent/received to the broker. Default is 131072. Minimum is 4096 and max is any large value (representable by an int). When receiving packets, this sets an internal buffer size in FFmpeg. It should be equal to or greater than the size of the published packets to the broker. Otherwise the received message may be truncated causing decoding errors.

connection_timeout
The timeout in seconds during the initial connection to the broker. The default value is rw_timeout, or 5 seconds if rw_timeout is not set.

delivery_mode *mode*
Sets the delivery mode of each message sent to broker. The following values are accepted:

persistent
Delivery mode set to ‘‘persistent’’ (2). This is the default value. Messages may be written to the broker’s disk depending on its setup.

non-persistent
Delivery mode set to ‘‘non-persistent’’ (1). Messages will stay in broker’s memory unless the broker is under memory pressure.

async
Asynchronous data filling wrapper for input stream.

Fill data in a background thread, to decouple I/O operation from demux thread.

```
async:<URL>
async:http://host/resource
async:cache:http://host/resource
```

bluray
Read BluRay playlist.

The accepted options are:

angle
BluRay angle

chapter
Start chapter (1...N)

playlist
Playlist to read (BDMV/PLAYLIST/?????.mpls)

Examples:

Read longest playlist from BluRay mounted to /mnt/bluray:

```
bluray:/mnt/bluray
```

Read angle 2 of playlist 4 from BluRay mounted to /mnt/bluray, start from chapter 2:

```
-playlist 4 -angle 2 -chapter 2 bluray:/mnt/bluray
```

cache
Caching wrapper for input stream.

Cache the input stream to temporary file. It brings seeking capability to live streams.

The accepted options are:

read_ahead_limit
Amount in bytes that may be read ahead when seeking isn't supported. Range is −1 to INT_MAX. −1 for unlimited. Default is 65536.

URL Syntax is

```
cache:<URL>
```

concat
Physical concatenation protocol.

Read and seek from many resources in sequence as if they were a unique resource.

A URL accepted by this protocol has the syntax:

```
concat:<URL1>|<URL2>|...|<URLN>
```

where *URL1*, *URL2*, ..., *URLN* are the urls of the resource to be concatenated, each one possibly specifying a distinct protocol.

For example to read a sequence of files *split1.mpeg*, *split2.mpeg*, *split3.mpeg* with **ffplay** use the command:

```
ffplay concat:split1.mpeg\|split2.mpeg\|split3.mpeg
```

Note that you may need to escape the character "|" which is special for many shells.

crypto
AES-encrypted stream reading protocol.

The accepted options are:

key Set the AES decryption key binary block from given hexadecimal representation.

iv Set the AES decryption initialization vector binary block from given hexadecimal representation.

Accepted URL formats:

```
crypto:<URL>
crypto+<URL>
```

data
Data in-line in the URI. See <**http://en.wikipedia.org/wiki/Data_URI_scheme**>.

For example, to convert a GIF file given inline with **ffmpeg**:

```
ffmpeg -i "
```

file
File access protocol.

Read from or write to a file.

A file URL can have the form:

```
file:<filename>
```

where *filename* is the path of the file to read.

An URL that does not have a protocol prefix will be assumed to be a file URL. Depending on the build, an URL that looks like a Windows path with the drive letter at the beginning will also be assumed to be a file URL (usually not the case in builds for unix-like systems).

For example to read from a file *input.mpeg* with **ffmpeg** use the command:

```
ffmpeg -i file:input.mpeg output.mpeg
```

This protocol accepts the following options:

truncate
Truncate existing files on write, if set to 1. A value of 0 prevents truncating. Default value is 1.

blocksize
Set I/O operation maximum block size, in bytes. Default value is `INT_MAX`, which results in not limiting the requested block size. Setting this value reasonably low improves user termination request reaction time, which is valuable for files on slow medium.

follow
If set to 1, the protocol will retry reading at the end of the file, allowing reading files that still are being written. In order for this to terminate, you either need to use the rw_timeout option, or use the interrupt callback (for API users).

seekable
Controls if seekability is advertised on the file. 0 means non-seekable, −1 means auto (seekable for normal files, non-seekable for named pipes).

Many demuxers handle seekable and non-seekable resources differently, overriding this might speed up opening certain files at the cost of losing some features (e.g. accurate seeking).

ftp
FTP (File Transfer Protocol).

Read from or write to remote resources using FTP protocol.

Following syntax is required.

```
ftp://[user[:password]@]server[:port]/path/to/remote/resource.mpeg
```

This protocol accepts the following options.

timeout
Set timeout in microseconds of socket I/O operations used by the underlying low level operation. By default it is set to −1, which means that the timeout is not specified.

ftp-user
Set a user to be used for authenticating to the FTP server. This is overridden by the user in the FTP URL.

ftp-password
Set a password to be used for authenticating to the FTP server. This is overridden by the password in the FTP URL, or by **ftp-anonymous-password** if no user is set.

ftp-anonymous-password
Password used when login as anonymous user. Typically an e−mail address should be used.

ftp-write-seekable
Control seekability of connection during encoding. If set to 1 the resource is supposed to be seekable, if set to 0 it is assumed not to be seekable. Default value is 0.

NOTE: Protocol can be used as output, but it is recommended to not do it, unless special care is taken (tests, customized server configuration etc.). Different FTP servers behave in different way during seek operation. ff* tools may produce incomplete content due to server limitations.

gopher
Gopher protocol.

gophers
Gophers protocol.

The Gopher protocol with TLS encapsulation.

hls
Read Apple HTTP Live Streaming compliant segmented stream as a uniform one. The M3U8 playlists describing the segments can be remote HTTP resources or local files, accessed using the standard file protocol. The nested protocol is declared by specifying "+*proto*" after the hls URI scheme name, where *proto* is either ‘‘file’’ or ‘‘http’’.

```
hls+http://host/path/to/remote/resource.m3u8
hls+file://path/to/local/resource.m3u8
```

Using this protocol is discouraged − the hls demuxer should work just as well (if not, please report the issues) and is more complete. To use the hls demuxer instead, simply use the direct URLs to the m3u8 files.

http

HTTP (Hyper Text Transfer Protocol).

This protocol accepts the following options:

seekable
Control seekability of connection. If set to 1 the resource is supposed to be seekable, if set to 0 it is assumed not to be seekable, if set to −1 it will try to autodetect if it is seekable. Default value is −1.

chunked_post
If set to 1 use chunked Transfer-Encoding for posts, default is 1.

content_type
Set a specific content type for the POST messages or for listen mode.

http_proxy
set HTTP proxy to tunnel through e.g. http://example.com:1234

headers
Set custom HTTP headers, can override built in default headers. The value must be a string encoding the headers.

multiple_requests
Use persistent connections if set to 1, default is 0.

post_data
Set custom HTTP post data.

referer
Set the Referer header. Include ’Referer: URL’ header in HTTP request.

user_agent
Override the User-Agent header. If not specified the protocol will use a string describing the libavformat build. (“Lavf/<version>”)

user-agent
This is a deprecated option, you can use user_agent instead it.

reconnect_at_eof
If set then eof is treated like an error and causes reconnection, this is useful for live / endless streams.

reconnect_streamed
If set then even streamed/non seekable streams will be reconnected on errors.

reconnect_on_network_error
Reconnect automatically in case of TCP/TLS errors during connect.

reconnect_on_http_error
A comma separated list of HTTP status codes to reconnect on. The list can include specific status codes (e.g. ’503’) or the strings ’4xx’ / ’5xx’.

reconnect_delay_max
Sets the maximum delay in seconds after which to give up reconnecting

mime_type
Export the MIME type.

http_version
Exports the HTTP response version number. Usually ‘‘1.0’’ or ‘‘1.1’’.

icy If set to 1 request ICY (SHOUTcast) metadata from the server. If the server supports this, the metadata has to be retrieved by the application by reading the **icy_metadata_headers** and **icy_metadata_packet** options. The default is 1.

icy_metadata_headers
If the server supports ICY metadata, this contains the ICY-specific HTTP reply headers, separated by newline characters.

icy_metadata_packet
If the server supports ICY metadata, and **icy** was set to 1, this contains the last non-empty metadata packet sent by the server. It should be polled in regular intervals by applications interested in mid-stream metadata updates.

cookies
Set the cookies to be sent in future requests. The format of each cookie is the same as the value of a Set-Cookie HTTP response field. Multiple cookies can be delimited by a newline character.

offset
Set initial byte offset.

end_offset
Try to limit the request to bytes preceding this offset.

method
When used as a client option it sets the HTTP method for the request.

When used as a server option it sets the HTTP method that is going to be expected from the client(s). If the expected and the received HTTP method do not match the client will be given a Bad Request response. When unset the HTTP method is not checked for now. This will be replaced by autodetection in the future.

listen
If set to 1 enables experimental HTTP server. This can be used to send data when used as an output option, or read data from a client with HTTP POST when used as an input option. If set to 2 enables experimental multi-client HTTP server. This is not yet implemented in ffmpeg.c and thus must not be used as a command line option.

```
# Server side (sending):
ffmpeg −i somefile.ogg −c copy −listen 1 −f ogg http://<server>:<port>

# Client side (receiving):
ffmpeg −i http://<server>:<port> −c copy somefile.ogg

# Client can also be done with wget:
wget http://<server>:<port> −O somefile.ogg

# Server side (receiving):
ffmpeg −listen 1 −i http://<server>:<port> −c copy somefile.ogg

# Client side (sending):
ffmpeg −i somefile.ogg −chunked_post 0 −c copy −f ogg http://<server>:

# Client can also be done with wget:
wget −−post−file=somefile.ogg http://<server>:<port>
```

send_expect_100
Send an Expect: 100−continue header for POST. If set to 1 it will send, if set to 0 it won’t, if set to −1 it will try to send if it is applicable. Default value is −1.

auth_type
Set HTTP authentication type. No option for Digest, since this method requires getting nonce parameters from the server first and can't be used straight away like Basic.

none
Choose the HTTP authentication type automatically. This is the default.

basic
Choose the HTTP basic authentication.

Basic authentication sends a Base64−encoded string that contains a user name and password for the client. Base64 is not a form of encryption and should be considered the same as sending the user name and password in clear text (Base64 is a reversible encoding). If a resource needs to be protected, strongly consider using an authentication scheme other than basic authentication. HTTPS/TLS should be used with basic authentication. Without these additional security enhancements, basic authentication should not be used to protect sensitive or valuable information.

HTTP Cookies

Some HTTP requests will be denied unless cookie values are passed in with the request. The **cookies** option allows these cookies to be specified. At the very least, each cookie must specify a value along with a path and domain. HTTP requests that match both the domain and path will automatically include the cookie value in the HTTP Cookie header field. Multiple cookies can be delimited by a newline.

The required syntax to play a stream specifying a cookie is:

```
ffplay −cookies "nlqptid=nltid=tsn; path=/; domain=somedomain.com;" http:
```

Icecast
Icecast protocol (stream to Icecast servers)

This protocol accepts the following options:

ice_genre
Set the stream genre.

ice_name
Set the stream name.

ice_description
Set the stream description.

ice_url
Set the stream website URL.

ice_public
Set if the stream should be public. The default is 0 (not public).

user_agent
Override the User-Agent header. If not specified a string of the form ‘‘Lavf/<version>’’ will be used.

password
Set the Icecast mountpoint password.

content_type
Set the stream content type. This must be set if it is different from audio/mpeg.

legacy_icecast
This enables support for Icecast versions < 2.4.0, that do not support the HTTP PUT method but the SOURCE method.

tls Establish a TLS (HTTPS) connection to Icecast.

```
icecast://[<username>[:<password>]@]<server>:<port>/<mountpoint>
```

mmst
MMS (Microsoft Media Server) protocol over TCP.

mmsh
MMS (Microsoft Media Server) protocol over HTTP.

The required syntax is:

```
mmsh://<server>[:<port>][/<app>][/<playpath>]
```

md5
MD5 output protocol.

Computes the MD5 hash of the data to be written, and on close writes this to the designated output or stdout if none is specified. It can be used to test muxers without writing an actual file.

Some examples follow.

```
# Write the MD5 hash of the encoded AVI file to the file output.avi.md5.
ffmpeg -i input.flv -f avi -y md5:output.avi.md5

# Write the MD5 hash of the encoded AVI file to stdout.
ffmpeg -i input.flv -f avi -y md5:
```

Note that some formats (typically MOV) require the output protocol to be seekable, so they will fail with the MD5 output protocol.

pipe
UNIX pipe access protocol.

Read and write from UNIX pipes.

The accepted syntax is:

```
pipe:[<number>]
```

number is the number corresponding to the file descriptor of the pipe (e.g. 0 for stdin, 1 for stdout, 2 for stderr). If*number* is not specified, by def ault the stdout file descriptor will be used for writing, stdin for reading.

For example to read from stdin with **ffmpeg**:

```
cat test.wav | ffmpeg -i pipe:0
# ...this is the same as...
cat test.wav | ffmpeg -i pipe:
```

For writing to stdout with **ffmpeg**:

```
ffmpeg -i test.wav -f avi pipe:1 | cat > test.avi
# ...this is the same as...
ffmpeg -i test.wav -f avi pipe: | cat > test.avi
```

This protocol accepts the following options:

blocksize
Set I/O operation maximum block size, in bytes. Default value is `INT_MAX`, which results in not limiting the requested block size. Setting this value reasonably low improves user termination request reaction time, which is valuable if data transmission is slow.

Note that some formats (typically MOV), require the output protocol to be seekable, so they will fail with the pipe output protocol.

prompeg
Pro-MPEG Code of Practice #3 Release 2 FEC protocol.

The Pro-MPEG CoP#3 FEC is a 2D parity-check forward error correction mechanism for MPEG-2 Transport Streams sent over RTP.

This protocol must be used in conjunction with the `rtp_mpegts` muxer and the `rtp` protocol.

The required syntax is:

```
-f rtp_mpegts -fec prompeg=<option>=<val>... rtp://<hostname>:<port>
```

The destination UDP ports are `port + 2` for the column FEC stream and `port + 4` for the row FEC stream.

This protocol accepts the following options:

l=*n* The number of columns (4−20, LxD <= 100)

d=*n*
The number of rows (4−20, LxD <= 100)

Example usage:

```
-f rtp_mpegts -fec prompeg=l=8:d=4 rtp://<hostname>:<port>
```

rist

Reliable Internet Streaming Transport protocol

The accepted options are:

rist_profile
Supported values:

simple
main
This one is default.

advanced

buffer_size
Set internal RIST buffer size in milliseconds for retransmission of data. Default value is 0 which means the librist default (1 sec). Maximum value is 30 seconds.

pkt_size
Set maximum packet size for sending data. 1316 by default.

log_level
Set loglevel for RIST logging messages. You only need to set this if you explicitly want to enable debug level messages or packet loss simulation, otherwise the regular loglevel is respected.

secret
Set override of encryption secret, by default is unset.

encryption
Set encryption type, by default is disabled. Acceptable values are 128 and 256.

rtmp

Real-Time Messaging Protocol.

The Real-Time Messaging Protocol (RTMP) is used for streaming multimedia content across a TCP/IP network.

The required syntax is:

```
rtmp://[<username>:<password>@]<server>[:<port>][/<app>][/<instance>][/<p
```

The accepted parameters are:

username
An optional username (mostly for publishing).

password
An optional password (mostly for publishing).

server
The address of the RTMP server.

port
The number of the TCP port to use (by default is 1935).

app
It is the name of the application to access. It usually corresponds to the path where the application is installed on the RTMP server (e.g. */ondemand/*, */flash/live/*, etc.). You can override the value parsed from the URI through the `rtmp_app` option, too.

playpath
It is the path or name of the resource to play with reference to the application specified in *app*, may be prefixed by ‘‘mp4:’’. You can override the value parsed from the URI through the `rtmp_playpath` option, too.

listen
Act as a server, listening for an incoming connection.

timeout
Maximum time to wait for the incoming connection. Implies listen.

Additionally, the following parameters can be set via command line options (or in code via `AVOption`s):

rtmp_app
Name of application to connect on the RTMP server. This option overrides the parameter specified in the URI.

rtmp_buffer
Set the client buffer time in milliseconds. The default is 3000.

rtmp_conn
Extra arbitrary AMF connection parameters, parsed from a string, e.g. like `B:1 S:authMe O:1 NN:code:1.23 NS:flag:ok O:0`. Each value is prefixed by a single character denoting the type, B for Boolean, N for number, S for string, O for object, or Z for null, followed by a colon. For Booleans the data must be either 0 or 1 for FALSE or TRUE, respectively. Likewise for Objects the data must be 0 or 1 to end or begin an object, respectively. Data items in subobjects may be named, by prefixing the type with ’N’ and specifying the name before the value (i.e. `NB:myFlag:1`). This option may be used multiple times to construct arbitrary AMF sequences.

rtmp_flashver
Version of the Flash plugin used to run the SWF player. The default is LNX 9,0,124,2. (When publishing, the default is FMLE/3.0 (compatible; <libavformat version>).)

rtmp_flush_interval
Number of packets flushed in the same request (RTMPT only). The default is 10.

rtmp_live
Specify that the media is a live stream. No resuming or seeking in live streams is possible. The default value is `any`, which means the subscriber first tries to play the live stream specified in the playpath. If a live stream of that name is not found, it plays the recorded stream. The other possible values are `live` and `recorded`.

rtmp_pageurl
URL of the web page in which the media was embedded. By default no value will be sent.

rtmp_playpath
Stream identifier to play or to publish. This option overrides the parameter specified in the URI.

rtmp_subscribe
Name of live stream to subscribe to. By default no value will be sent. It is only sent if the option is specified or if rtmp_live is set to live.

rtmp_swfhash
SHA256 hash of the decompressed SWF file (32 bytes).

rtmp_swfsize
Size of the decompressed SWF file, required for SWFVerification.

rtmp_swfurl
URL of the SWF player for the media. By default no value will be sent.

rtmp_swfverify
URL to player swf file, compute hash/size automatically.

rtmp_tcurl
URL of the target stream. Defaults to proto://host[:port]/app.

For example to read with **ffplay** a multimedia resource named ‘‘sample’’ from the application ‘‘vod’’ from an RTMP server ‘‘myserver’’:

```
ffplay rtmp://myserver/vod/sample
```

To publish to a password protected server, passing the playpath and app names separately:

```
ffmpeg -re -i <input> -f flv -rtmp_playpath some/long/path -rtmp_app long
```

rtmpe
Encrypted Real-Time Messaging Protocol.

The Encrypted Real-Time Messaging Protocol (RTMPE) is used for streaming multimedia content within standard cryptographic primitives, consisting of Diffie-Hellman key exchange and HMACSHA256, generating a pair of RC4 keys.

rtmps
Real-Time Messaging Protocol over a secure SSL connection.

The Real-Time Messaging Protocol (RTMPS) is used for streaming multimedia content across an encrypted connection.

rtmpt
Real-Time Messaging Protocol tunneled through HTTP.

The Real-Time Messaging Protocol tunneled through HTTP (RTMPT) is used for streaming multimedia content within HTTP requests to traverse firewalls.

rtmpte
Encrypted Real-Time Messaging Protocol tunneled through HTTP.

The Encrypted Real-Time Messaging Protocol tunneled through HTTP (RTMPTE) is used for streaming multimedia content within HTTP requests to traverse firewalls.

rtmpts
Real-Time Messaging Protocol tunneled through HTTPS.

The Real-Time Messaging Protocol tunneled through HTTPS (RTMPTS) is used for streaming multimedia content within HTTPS requests to traverse firewalls.

libsmbclient
libsmbclient permits one to manipulate CIFS/SMB network resources.

Following syntax is required.

```
smb://[[domain:]user[:password@]]server[/share[/path[/file]]]
```

This protocol accepts the following options.

timeout
Set timeout in milliseconds of socket I/O operations used by the underlying low level operation. By default it is set to −1, which means that the timeout is not specified.

truncate
Truncate existing files on write, if set to 1. A value of 0 prevents truncating. Default value is 1.

workgroup
Set the workgroup used for making connections. By default workgroup is not specified.

For more information see: <**http://www.samba.org/**>.

libssh
Secure File Transfer Protocol via libssh

Read from or write to remote resources using SFTP protocol.

Following syntax is required.

```
sftp://[user[:password]@]server[:port]/path/to/remote/resource.mpeg
```

This protocol accepts the following options.

timeout
Set timeout of socket I/O operations used by the underlying low level operation. By default it is set to −1, which means that the timeout is not specified.

truncate
Truncate existing files on write, if set to 1. A value of 0 prevents truncating. Default value is 1.

private_key
Specify the path of the file containing private key to use during authorization. By default libssh searches for keys in the *~/.ssh/* directory.

Example: Play a file stored on remote server.

```
ffplay sftp://user:password@server_address:22/home/user/resource.mpeg
```

librtmp rtmp, rtmpe, rtmps, rtmpt, rtmpte
Real-Time Messaging Protocol and its variants supported through librtmp.

Requires the presence of the librtmp headers and library during configuration. You need to explicitly configure the build with ‘‘−−enable−librtmp’’. If enabled this will replace the native RTMP protocol.

This protocol provides most client functions and a few server functions needed to support RTMP, RTMP tunneled in HTTP (RTMPT), encrypted RTMP (RTMPE), RTMP over SSL/TLS (RTMPS) and tunneled variants of these encrypted types (RTMPTE, RTMPTS).

The required syntax is:

```
<rtmp_proto>://<server>[:<port>][/<app>][/<playpath>] <options>
```

where *rtmp_proto* is one of the strings ‘‘rtmp’’, ‘‘rtmpt’’, ‘‘rtmpe’’, ‘‘rtmps’’, ‘‘rtmpte’’, ‘‘rtmpts’’ corresponding to each RTMP variant, and *server*, *port*, *app* and *playpath* have the same meaning as specified for the RTMP native protocol. *options* contains a list of space-separated options of the form *key=val*.

See the librtmp manual page (man 3 librtmp) for more information.

For example, to stream a file in real-time to an RTMP server using **ffmpeg**:

```
ffmpeg −re −i myfile −f flv rtmp://myserver/live/mystream
```

To play the same stream using **ffplay**:

```
ffplay "rtmp://myserver/live/mystream live=1"
```

rtp
Real-time Transport Protocol.

The required syntax for an RTP URL is: rtp://*hostname*[:*port*][?*option=val*...]

port specifies the RTP port to use.

The following URL options are supported:

ttl=*n*
Set the TTL (Time-To-Live) value (for multicast only).

rtcpport=*n*
Set the remote RTCP port to *n*.

localrtpport=*n*
Set the local RTP port to *n*.

localrtcpport=*n*’
Set the local RTCP port to *n*.

pkt_size=*n*
Set max packet size (in bytes) to *n*.

buffer_size=*size*
Set the maximum UDP socket buffer size in bytes.

connect=0|1
Do a `connect()` on the UDP socket (if set to 1) or not (if set to 0).

sources=*ip*[**,***ip*]
List allowed source IP addresses.

block=*ip*[**,***ip*]
List disallowed (blocked) source IP addresses.

write_to_source=0|1
Send packets to the source address of the latest received packet (if set to 1) or to a default remote address (if set to 0).

localport=*n*
Set the local RTP port to *n*.

timeout=*n*
Set timeout (in microseconds) of socket I/O operations to *n*.

This is a deprecated option. Instead, **localrtpport** should be used.

Important notes:

1. If **rtcpport** is not set the RTCP port will be set to the RTP port value plus 1.
2. If **localrtpport** (the local RTP port) is not set any available port will be used for the local RTP and RTCP ports.
3. If **localrtcpport** (the local RTCP port) is not set it will be set to the local RTP port value plus 1.

rtsp
Real-Time Streaming Protocol.

RTSP is not technically a protocol handler in libavformat, it is a demuxer and muxer. The demuxer supports both normal RTSP (with data transferred over RTP; this is used by e.g. Apple and Microsoft) and Real-RTSP (with data transferred over RDT).

The muxer can be used to send a stream using RTSP ANNOUNCE to a server supporting it (currently Darwin Streaming Server and Mischa Spiegelmock’s <**https://github.com/revmischa/rtsp−server**>).

The required syntax for a RTSP url is:

```
rtsp://<hostname>[:<port>]/<path>
```

Options can be set on the **ffmpeg**/**ffplay** command line, or set in code via `AVOption`s or in `avformat_open_input`.

The following options are supported.

initial_pause
Do not start playing the stream immediately if set to 1. Default value is 0.

rtsp_transport
Set RTSP transport protocols.

It accepts the following values:

udp
Use UDP as lower transport protocol.

tcp Use TCP (interleaving within the RTSP control channel) as lower transport protocol.

udp_multicast
Use UDP multicast as lower transport protocol.

http
Use HTTP tunneling as lower transport protocol, which is useful for passing proxies.

Multiple lower transport protocols may be specified, in that case they are tried one at a time (if the setup of one fails, the next one is tried). For the muxer, only the **tcp** and **udp** options are supported.

rtsp_flags
Set RTSP flags.

The following values are accepted:

filter_src
Accept packets only from negotiated peer address and port.

listen
Act as a server, listening for an incoming connection.

prefer_tcp
Try TCP for RTP transport first, if TCP is available as RTSP RTP transport.

Default value is **none**.

allowed_media_types
Set media types to accept from the server.

The following flags are accepted:

video
audio
data

By default it accepts all media types.

min_port
Set minimum local UDP port. Default value is 5000.

max_port
Set maximum local UDP port. Default value is 65000.

timeout
Set maximum timeout (in seconds) to wait for incoming connections.

A value of −1 means infinite (default). This option implies the **rtsp_flags** set to **listen**.

reorder_queue_size
Set number of packets to buffer for handling of reordered packets.

stimeout
Set socket TCP I/O timeout in microseconds.

user-agent
Override User-Agent header. If not specified, it defaults to the libavformat identifier string.

When receiving data over UDP, the demuxer tries to reorder received packets (since they may arrive out of order, or packets may get lost totally). This can be disabled by setting the maximum demuxing delay to zero (via the `max_delay` field of AVFormatContext).

When watching multi-bitrate Real-RTSP streams with **ffplay**, the streams to display can be chosen with `-vst` *n* and `-ast` *n* for video and audio respectively, and can be switched on the fly by pressing `v` and `a`.

Examples

The following examples all make use of the **ffplay** and **ffmpeg** tools.

- Watch a stream over UDP, with a max reordering delay of 0.5 seconds:

```
ffplay -max_delay 500000 -rtsp_transport udp rtsp://server/video.mp4
```

- Watch a stream tunneled over HTTP:

```
ffplay -rtsp_transport http rtsp://server/video.mp4
```

- Send a stream in realtime to a RTSP server, for others to watch:

```
ffmpeg -re -i <input> -f rtsp -muxdelay 0.1 rtsp://server/live.sdp
```

- Receive a stream in realtime:

```
ffmpeg -rtsp_flags listen -i rtsp://ownaddress/live.sdp <output>
```

sap
Session Announcement Protocol (RFC 2974). This is not technically a protocol handler in libavformat, it is a muxer and demuxer. It is used for signalling of RTP streams, by announcing the SDP for the streams regularly on a separate port.

Muxer

The syntax for a SAP url given to the muxer is:

```
sap://<destination>[:<port>][?<options>]
```

The RTP packets are sent to *destination* on port *port*, or to port 5004 if no port is specified. *options* is a `&`-separated list. The following options are supported:

announce_addr=*address*
Specify the destination IP address for sending the announcements to. If omitted, the announcements are sent to the commonly used SAP announcement multicast address 224.2.127.254 (sap.mcast.net), or ff0e::2:7ffe if *destination* is an IPv6 address.

announce_port=*port*
Specify the port to send the announcements on, defaults to 9875 if not specified.

ttl=*ttl*
Specify the time to live value for the announcements and RTP packets, defaults to 255.

same_port=*0|1*
If set to 1, send all RTP streams on the same port pair. If zero (the default), all streams are sent on unique ports, with each stream on a port 2 numbers higher than the previous. VLC/Live555 requires this to be set to 1, to be able to receive the stream. The RTP stack in libavformat for receiving requires all streams to be sent on unique ports.

Example command lines follow.

To broadcast a stream on the local subnet, for watching in VLC:

```
ffmpeg -re -i <input> -f sap sap://224.0.0.255?same_port=1
```

Similarly, for watching in **ffplay**:

```
ffmpeg -re -i <input> -f sap sap://224.0.0.255
```

And for watching in **ffplay**, over IPv6:

```
ffmpeg -re -i <input> -f sap sap://[ff0e::1:2:3:4]
```

Demuxer

The syntax for a SAP url given to the demuxer is:

```
sap://[<address>][:<port>]
```

address is the multicast address to listen for announcements on, if omitted, the default 224.2.127.254 (sap.mcast.net) is used. *port* is the port that is listened on, 9875 if omitted.

The demuxers listens for announcements on the given address and port. Once an announcement is received, it tries to receive that particular stream.

Example command lines follow.

To play back the first stream announced on the normal SAP multicast address:

```
ffplay sap://
```

To play back the first stream announced on one the default IPv6 SAP multicast address:

```
ffplay sap://[ff0e::2:7ffe]
```

sctp

Stream Control Transmission Protocol.

The accepted URL syntax is:

```
sctp://<host>:<port>[?<options>]
```

The protocol accepts the following options:

listen
If set to any value, listen for an incoming connection. Outgoing connection is done by default.

max_streams
Set the maximum number of streams. By default no limit is set.

srt

Haivision Secure Reliable Transport Protocol via libsrt.

The supported syntax for a SRT URL is:

```
srt://<hostname>:<port>[?<options>]
```

options contains a list of &−separated options of the form *key=val*.

or

```
<options> srt://<hostname>:<port>
```

options contains a list of ’−*key val*’ options.

This protocol accepts the following options.

connect_timeout=*milliseconds*
Connection timeout; SRT cannot connect for RTT > 1500 msec (2 handshake exchanges) with the default connect timeout of 3 seconds. This option applies to the caller and rendezvous connection modes. The connect timeout is 10 times the value set for the rendezvous mode (which can be used as a workaround for this connection problem with earlier versions).

ffs=*bytes*
Flight Flag Size (Window Size), in bytes. FFS is actually an internal parameter and you should set it to not less than **recv_buffer_size** and **mss**. The default value is relatively large, therefore unless you set a very large receiver buffer, you do not need to change this option. Default value is 25600.

inputbw=*bytes/seconds*
Sender nominal input rate, in bytes per seconds. Used along with **oheadbw**, when **maxbw** is set to relative (0), to calculate maximum sending rate when recovery packets are sent along with the main media stream: **inputbw** * (100 + **oheadbw**) / 100 if **inputbw** is not set while **maxbw** is set to relative (0), the actual input rate is evaluated inside the library. Default value is 0.

iptos=*tos*
IP Type of Service. Applies to sender only. Default value is 0xB8.

ipttl=*ttl*
IP Time To Live. Applies to sender only. Default value is 64.

latency=*microseconds*
Timestamp-based Packet Delivery Delay. Used to absorb bursts of missed packet retransmissions. This flag sets both **rcvlatency** and **peerlatency** to the same value. Note that prior to version 1.3.0 this is the only flag to set the latency, however this is effectively equivalent to setting **peerlatency**, when side is sender and **rcvlatency** when side is receiver, and the bidirectional stream sending is not supported.

listen_timeout=*microseconds*
Set socket listen timeout.

maxbw=*bytes/seconds*
Maximum sending bandwidth, in bytes per seconds. −1 infinite (CSRTCC limit is 30mbps) 0 relative to input rate (see **inputbw**) >0 absolute limit value Default value is 0 (relative)

mode=*caller|listener|rendezvous*
Connection mode. **caller** opens client connection. **listener** starts serv er to listen for incoming connections. **rendezvous** use Rendez-Vous connection mode. Default value is caller.

mss=*bytes*
Maximum Segment Size, in bytes. Used for buffer allocation and rate calculation using a packet counter assuming fully filled packets. The smallest MSS between the peers is used. This is 1500 by default in the overall internet. This is the maximum size of the UDP packet and can be only decreased, unless you have some unusual dedicated network settings. Default value is 1500.

nakreport=*1|0*
If set to 1, Receiver will send ‘UMSG_LOSSREPORT‘ messages periodically until a lost packet is retransmitted or intentionally dropped. Default value is 1.

oheadbw=*percents*
Recovery bandwidth overhead above input rate, in percents. See **inputbw**. Default value is 25%.

passphrase=*string*
HaiCrypt Encryption/Decryption Passphrase string, length from 10 to 79 characters. The passphrase is the shared secret between the sender and the receiver. It is used to generate the Key Encrypting Key using PBKDF2 (Password-Based Key Derivation Function). It is used only if **pbkeylen** is non-zero. It is used on the receiver only if the received data is encrypted. The configured passphrase cannot be recovered (write-only).

enforced_encryption=*1|0*
If true, both connection parties must have the same password set (including empty, that is, with no encryption). If the password doesn’t match or only one side is unencrypted, the connection is rejected. Default is true.

kmrefreshrate=*packets*
The number of packets to be transmitted after which the encryption key is switched to a new key. Default is −1. −1 means auto (0x1000000 in srt library). The range for this option is integers in the 0 − `INT_MAX`.

kmpreannounce=*packets*
The interval between when a new encryption key is sent and when switchover occurs. This value also applies to the subsequent interval between when switchover occurs and when the old encryption key is decommissioned. Default is −1. −1 means auto (0x1000 in srt library). The range for this option is integers in the 0 − `INT_MAX`.

payload_size=*bytes*
Sets the maximum declared size of a packet transferred during the single call to the sending function in Live mode. Use 0 if this value isn't used (which is default in file mode). Default is −1 (automatic), which typically means MPEG-TS; if you are going to use SRT to send any different kind of payload, such as, for example, wrapping a live stream in very small frames, then you can use a bigger maximum frame size, though not greater than 1456 bytes.

pkt_size=*bytes*
Alias for **payload_size**.

peerlatency=*microseconds*
The latency value (as described in **rcvlatency**) that is set by the sender side as a minimum value for the receiver.

pbkeylen=*bytes*
Sender encryption key length, in bytes. Only can be set to 0, 16, 24 and 32. Enable sender encryption if not 0. Not required on receiver (set to 0), key size obtained from sender in HaiCrypt handshake. Default value is 0.

rcvlatency=*microseconds*
The time that should elapse since the moment when the packet was sent and the moment when it's delivered to the receiver application in the receiving function. This time should be a buffer time large enough to cover the time spent for sending, unexpectedly extended RTT time, and the time needed to retransmit the lost UDP packet. The effective latency value will be the maximum of this options' value and the value of **peerlatency** set by the peer side. Before version 1.3.0 this option is only available as **latency**.

recv_buffer_size=*bytes*
Set UDP receive buffer size, expressed in bytes.

send_buffer_size=*bytes*
Set UDP send buffer size, expressed in bytes.

timeout=*microseconds*
Set raise error timeouts for read, write and connect operations. Note that the SRT library has internal timeouts which can be controlled separately, the value set here is only a cap on those.

tlpktdrop=*1|0*
Too-late Packet Drop. When enabled on receiver, it skips missing packets that have not been delivered in time and delivers the following packets to the application when their time-to-play has come. It also sends a fake ACK to the sender. When enabled on sender and enabled on the receiving peer, the sender drops the older packets that have no chance of being delivered in time. It was automatically enabled in the sender if the receiver supports it.

sndbuf=*bytes*
Set send buffer size, expressed in bytes.

rcvbuf=*bytes*
Set receive buffer size, expressed in bytes.

Receive buffer must not be greater than **ffs**.

lossmaxttl=*packets*
The value up to which the Reorder Tolerance may grow. When Reorder Tolerance is > 0, then packet loss report is delayed until that number of packets come in. Reorder Tolerance increases every time a ''belated'' packet has come, but it wasn't due to retransmission (that is, when UDP packets tend to

come out of order), with the difference between the latest sequence and this packet's sequence, and not more than the value of this option. By default it's 0, which means that this mechanism is turned off, and the loss report is always sent immediately upon experiencing a ''gap'' in sequences.

minversion
The minimum SRT version that is required from the peer. A connection to a peer that does not satisfy the minimum version requirement will be rejected.

The version format in hex is 0xXXYYZZ for x.y.z in human readable form.

streamid=*string*
A string limited to 512 characters that can be set on the socket prior to connecting. This stream ID will be able to be retrieved by the listener side from the socket that is returned from srt_accept and was connected by a socket with that set stream ID. SRT does not enforce any special interpretation of the contents of this string. This option doesnXt make sense in Rendezvous connection; the result might be that simply one side will override the value from the other side and itXs the matter of luck which one would win

smoother=*live|file*
The type of Smoother used for the transmission for that socket, which is responsible for the transmission and congestion control. The Smoother type must be exactly the same on both connecting parties, otherwise the connection is rejected.

messageapi=*1|0*
When set, this socket uses the Message API, otherwise it uses Buffer API. Note that in live mode (see **transtype**) thereXs only message API available. In File mode you can chose to use one of two modes:

Stream API (default, when this option is false). In this mode you may send as many data as you wish with one sending instruction, or even use dedicated functions that read directly from a file. The internal facility will take care of any speed and congestion control. When receiving, you can also receive as many data as desired, the data not extracted will be waiting for the next call. There is no boundary between data portions in the Stream mode.

Message API. In this mode your single sending instruction passes exactly one piece of data that has boundaries (a message). Contrary to Live mode, this message may span across multiple UDP packets and the only size limitation is that it shall fit as a whole in the sending buffer. The receiver shall use as large buffer as necessary to receive the message, otherwise the message will not be given up. When the message is not complete (not all packets received or there was a packet loss) it will not be given up.

transtype=*live|file*
Sets the transmission type for the socket, in particular, setting this option sets multiple other parameters to their default values as required for a particular transmission type.

live: Set options as for live transmission. In this mode, you should send by one sending instruction only so many data that fit in one UDP packet, and limited to the value defined first in **payload_size** (1316 is default in this mode). There is no speed control in this mode, only the bandwidth control, if configured, in order to not exceed the bandwidth with the overhead transmission (retransmitted and control packets).

file: Set options as for non-live transmission. See **messageapi** for further explanations

linger=*seconds*
The number of seconds that the socket waits for unsent data when closing. Default is −1. −1 means auto (off with 0 seconds in live mode, on with 180 seconds in file mode). The range for this option is integers in the 0 − `INT_MAX`.

For more information see: <**https://github.com/Haivision/srt**>.

srtp
Secure Real-time Transport Protocol.

The accepted options are:

srtp_in_suite
srtp_out_suite
Select input and output encoding suites.

Supported values:

AES_CM_128_HMAC_SHA1_80
SRTP_AES128_CM_HMAC_SHA1_80
AES_CM_128_HMAC_SHA1_32
SRTP_AES128_CM_HMAC_SHA1_32

srtp_in_params
srtp_out_params
Set input and output encoding parameters, which are expressed by a base64−encoded representation of a binary block. The first 16 bytes of this binary block are used as master key, the following 14 bytes are used as master salt.

subfile
Virtually extract a segment of a file or another stream. The underlying stream must be seekable.

Accepted options:

start
Start offset of the extracted segment, in bytes.

end
End offset of the extracted segment, in bytes. If set to 0, extract till end of file.

Examples:

Extract a chapter from a DVD VOB file (start and end sectors obtained externally and multiplied by 2048):

```
subfile,,start,153391104,end,268142592,,:/media/dvd/VIDEO_TS/VTS_08_1.VOB
```

Play an AVI file directly from a TAR archive:

```
subfile,,start,183241728,end,366490624,,:archive.tar
```

Play a MPEG-TS file from start offset till end:

```
subfile,,start,32815239,end,0,,:video.ts
```

tee
Writes the output to multiple protocols. The individual outputs are separated by |

```
tee:file://path/to/local/this.avi|file://path/to/local/that.avi
```

tcp
Transmission Control Protocol.

The required syntax for a TCP url is:

```
tcp://<hostname>:<port>[?<options>]
```

options contains a list of &−separated options of the form *key=val*.

The list of supported options follows.

listen=*2|1|0*
Listen for an incoming connection. 0 disables listen, 1 enables listen in single client mode, 2 enables listen in multi-client mode. Default value is 0.

timeout=*microseconds*
Set raise error timeout, expressed in microseconds.

This option is only relevant in read mode: if no data arrived in more than this time interval, raise error.

listen_timeout=*milliseconds*
Set listen timeout, expressed in milliseconds.

recv_buffer_size=*bytes*
Set receive buffer size, expressed bytes.

send_buffer_size=*bytes*
Set send buffer size, expressed bytes.

tcp_nodelay=*1|0*
Set TCP_NODELAY to disable Nagle's algorithm. Default value is 0.

tcp_mss=*bytes*
Set maximum segment size for outgoing TCP packets, expressed in bytes.

The following example shows how to setup a listening TCP connection with **ffmpeg**, which is then accessed with **ffplay**:

```
ffmpeg -i <input> -f <format> tcp://<hostname>:<port>?listen
ffplay tcp://<hostname>:<port>
```

tls

Transport Layer Security (TLS) / Secure Sockets Layer (SSL)

The required syntax for a TLS/SSL url is:

```
tls://<hostname>:<port>[?<options>]
```

The following parameters can be set via command line options (or in code via `AVOption`s):

ca_file, cafile=*filename*
A file containing certificate authority (CA) root certificates to treat as trusted. If the linked TLS library contains a default this might not need to be specified for verification to work, but not all libraries and setups have defaults built in. The file must be in OpenSSL PEM format.

tls_verify=*1|0*
If enabled, try to verify the peer that we are communicating with. Note, if using OpenSSL, this currently only makes sure that the peer certificate is signed by one of the root certificates in the CA database, but it does not validate that the certificate actually matches the host name we are trying to connect to. (With other backends, the host name is validated as well.)

This is disabled by default since it requires a CA database to be provided by the caller in many cases.

cert_file, cert=*filename*
A file containing a certificate to use in the handshake with the peer. (When operating as server, in listen mode, this is more often required by the peer, while client certificates only are mandated in certain setups.)

key_file, key=*filename*
A file containing the private key for the certificate.

listen=*1|0*
If enabled, listen for connections on the provided port, and assume the server role in the handshake instead of the client role.

http_proxy
The HTTP proxy to tunnel through, e.g. `http://example.com:1234`. The proxy must support the CONNECT method.

Example command lines:

To create a TLS/SSL server that serves an input stream.

```
ffmpeg -i <input> -f <format> tls://<hostname>:<port>?listen&cert=<server
```

To play back a stream from the TLS/SSL server using **ffplay**:

```
ffplay tls://<hostname>:<port>
```

udp

User Datagram Protocol.

The required syntax for an UDP URL is:

```
udp://<hostname>:<port>[?<options>]
```

options contains a list of &−separated options of the form *key=val*.

In case threading is enabled on the system, a circular buffer is used to store the incoming data, which allows one to reduce loss of data due to UDP socket buffer overruns. The *fifo_size* and *overrun_nonfatal* options are related to this buffer.

The list of supported options follows.

buffer_size=*size*
Set the UDP maximum socket buffer size in bytes. This is used to set either the receive or send buffer size, depending on what the socket is used for. Default is 32 KB for output, 384 KB for input. See also *fifo_size*.

bitrate=*bitrate*
If set to nonzero, the output will have the specified constant bitrate if the input has enough packets to sustain it.

burst_bits=*bits*
When using *bitrate* this specifies the maximum number of bits in packet bursts.

localport=*port*
Override the local UDP port to bind with.

localaddr=*addr*
Local IP address of a network interface used for sending packets or joining multicast groups.

pkt_size=*size*
Set the size in bytes of UDP packets.

reuse=*1|0*
Explicitly allow or disallow reusing UDP sockets.

ttl=*ttl*
Set the time to live value (for multicast only).

connect=*1|0*
Initialize the UDP socket with `connect()`. In this case, the destination address can't be changed with ff_udp_set_remote_url later. If the destination address isn't known at the start, this option can be specified in ff_udp_set_remote_url, too. This allows finding out the source address for the packets with getsockname, and makes writes return with AVERROR(ECONNREFUSED) if ''destination unreachable'' is received. For receiving, this gives the benefit of only receiving packets from the specified peer address/port.

sources=*address*[**,***address*]
Only receive packets sent from the specified addresses. In case of multicast, also subscribe to multicast traffic coming from these addresses only.

block=*address*[**,***address*]
Ignore packets sent from the specified addresses. In case of multicast, also exclude the source addresses in the multicast subscription.

fifo_size=*units*
Set the UDP receiving circular buffer size, expressed as a number of packets with size of 188 bytes. If not specified defaults to 7*4096.

overrun_nonfatal=*1|0*
Survive in case of UDP receiving circular buffer overrun. Default value is 0.

timeout=*microseconds*
Set raise error timeout, expressed in microseconds.

This option is only relevant in read mode: if no data arrived in more than this time interval, raise error.

broadcast=*1|0*
Explicitly allow or disallow UDP broadcasting.

Note that broadcasting may not work properly on networks having a broadcast storm protection.

Examples

- Use **ffmpeg** to stream over UDP to a remote endpoint:

```
ffmpeg -i <input> -f <format> udp://<hostname>:<port>
```

- Use **ffmpeg** to stream in mpegts format over UDP using 188 sized UDP packets, using a large input buffer:

```
ffmpeg -i <input> -f mpegts udp://<hostname>:<port>?pkt_size=188&buffe
```

- Use **ffmpeg** to receive over UDP from a remote endpoint:

```
ffmpeg -i udp://[<multicast-address>]:<port> ...
```

unix
Unix local socket

The required syntax for a Unix socket URL is:

```
unix://<filepath>
```

The following parameters can be set via command line options (or in code via `AVOption`s):

timeout
Timeout in ms.

listen
Create the Unix socket in listening mode.

zmq
ZeroMQ asynchronous messaging using the libzmq library.

This library supports unicast streaming to multiple clients without relying on an external server.

The required syntax for streaming or connecting to a stream is:

```
zmq:tcp://ip-address:port
```

Example: Create a localhost stream on port 5555:

```
ffmpeg -re -i input -f mpegts zmq:tcp://127.0.0.1:5555
```

Multiple clients may connect to the stream using:

```
ffplay zmq:tcp://127.0.0.1:5555
```

Streaming to multiple clients is implemented using a ZeroMQ Pub-Sub pattern. The server side binds to a port and publishes data. Clients connect to the server (via IP address/port) and subscribe to the stream. The order in which the server and client start generally does not matter.

ffmpeg must be compiled with the --enable-libzmq option to support this protocol.

Options can be set on the **ffmpeg**/**ffplay** command line. The following options are supported:

pkt_size
Forces the maximum packet size for sending/receiving data. The default value is 131,072 bytes. On the server side, this sets the maximum size of sent packets via ZeroMQ. On the clients, it sets an internal buffer size for receiving packets. Note that pkt_size on the clients should be equal to or greater than pkt_size on the server. Otherwise the received message may be truncated causing decoding errors.

SEE ALSO

ffmpeg (1), **ffplay** (1), **ffprobe** (1), **libavformat** (3)

AUTHORS

The FFmpeg developers.

For details about the authorship, see the Git history of the project (git://source.ffmpeg.org/ffmpeg), e.g. by typing the command **git log** in the FFmpeg source directory, or browsing the online repository at <**http://source.ffmpeg.org**>.

Maintainers for the specific components are listed in the file *MAINTAINERS* in the source code tree.