Transport

Core protocol implementation

class **paramiko.transport.SecurityOptions**(*transport*)

Simple object containing the security preferences of an ssh transport. These are tuples of acceptable ciphers, digests, key types, and key exchange algorithms, listed in order of preference.

Changing the contents and/or order of these fields affects the underlying **Transport** (but only if you change them before starting the session). If you try to add an algorithm that paramiko doesn't recognize, `ValueError` will be raised. If you try to assign something besides a tuple to one of the fields, `TypeError` will be raised.

__repr__()

Returns a string representation of this object, for debugging.

ciphers

Symmetric encryption ciphers

compression ¶

Compression algorithms

digests

Digest (one-way hash) algorithms

kex

Key exchange algorithms

key_types

Public-key algorithms

class **paramiko.transport.Transport**(*sock*, *default_window_size=2097152*, *default_max_packet_size=32768*, *gss_kex=False*, *gss_deleg_creds=True*)

An SSH Transport attaches to a stream (usually a socket), negotiates an encrypted session, authenticates, and then creates stream tunnels, called **channels**, across the session. Multiple channels can be multiplexed across a single session (and often are, in the case of port forwardings).

Instances of this class may be used as context managers.

__init__(*sock*, *default_window_size=2097152*, *default_max_packet_size=32768*, *gss_kex=False*, *gss_deleg_creds=True*)

Create a new SSH session over an existing socket, or socket-like object. This only creates the **Transport** object; it doesn't begin the SSH session yet. Use **connect** or **start_client** to begin a client session, or **start_server** to begin a server session.

If the object is not actually a socket, it must have the following methods:

- `send(str)`: Writes from 1 to `len(str)` bytes, and returns an int representing the number of bytes written. Returns 0 or raises `EOFError` if the stream has been closed.
- `recv(int)`: Reads from 1 to `int` bytes and returns them as a string. Returns 0 or raises `EOFError` if the stream has been closed.
- `close()`: Closes the socket.
- `settimeout(n)`: Sets a (float) timeout on I/O operations.

For ease of use, you may also pass in an address (as a tuple) or a host string as the `sock` argument. (A host string is a hostname with an optional port (separated by `":"`) which will be converted into a tuple of `(hostname, port)`.) A socket will be connected to this address and used for communication. Exceptions from the `socket` call may be thrown in this case.

> **Note:**
> Modifying the the window and packet sizes might have adverse effects on your channels created from this transport. The default values are the same as in the OpenSSH code base and have been battle tested.

Parameters: • **sock** (*socket*) – a socket or socket-like object to create the session over.

- **default_window_size** (*int*) – sets the default window size on the transport. (defaults to 2097152)
- **default_max_packet_size** (*int*) – sets the default max packet size on the transport. (defaults to 32768)

Changed in version 1.15: Added the `default_window_size` and `default_max_packet_size` arguments.

`__repr__`()

Returns a string representation of this object, for debugging.

`accept`(*timeout=None*)

Return the next channel opened by the client over this transport, in server mode. If no channel is opened before the given timeout, `None` is returned.

Parameters: **timeout** (*int*) – seconds to wait for a channel, or `None` to wait forever
Returns: a new **`Channel`** opened by the client

`add_server_key`(*key*)

Add a host key to the list of keys used for server mode. When behaving as a server, the host key is used to sign certain packets during the SSH2 negotiation, so that the client can trust that we are who we say we are. Because this is used for signing, the key must contain private key info, not just the public half. Only one key of each type (RSA or DSS) is kept.

Parameters: **key** (*PKey*) – the host key to add, usually an **`RSAKey`** or **`DSSKey`**.

`atfork`()

Terminate this Transport without closing the session. On posix systems, if a Transport is open during process forking, both parent and child will share the underlying socket, but only one process can use the connection (without corrupting the session). Use this method to clean up a Transport object without disrupting the other process.

New in version 1.5.3.

`auth_gssapi_keyex`(*username*)

Authenticate to the server with GSS-API/SSPI if GSS-API kex is in use.

Parameters: **username** (*str*) – The username to authenticate as.
Returns: a list of auth types permissible for the next stage of authentication (normally empty)
Raises: **`BadAuthenticationType`** – if GSS-API Key Exchange was not performed (and no event was passed in)
Raises: **`AuthenticationException`** – if the authentication failed (and no event was passed in)
Raises: **`SSHException`** – if there was a network error

`auth_gssapi_with_mic`(*username*, *gss_host*, *gss_deleg_creds*)

Authenticate to the Server using GSS-API / SSPI.

Parameters:
- **username** (*str*) – The username to authenticate as
- **gss_host** (*str*) – The target host
- **gss_deleg_creds** (*bool*) – Delegate credentials or not

Returns: list of auth types permissible for the next stage of authentication (normally empty)

Raises: **`BadAuthenticationType`** – if gssapi-with-mic isn't allowed by the server (and no event was passed in)

Raises: **`AuthenticationException`** – if the authentication failed (and no event was passed in)

Raises: **`SSHException`** – if there was a network error

`auth_interactive`(*username*, *handler*, *submethods=''*)

Authenticate to the server interactively. A handler is used to answer arbitrary questions from the server. On many servers, this is just a dumb wrapper around PAM.

This method will block until the authentication succeeds or fails, peroidically calling the handler asynchronously to get answers to authentication questions. The handler may be called more than once if the server continues to ask questions.

The handler is expected to be a callable that will handle calls of the form: `handler(title, instructions, prompt_list)`. The `title` is meant to be a dialog-window title, and the `instructions` are user instructions (both are strings). `prompt_list` will be a list of prompts, each prompt being a tuple of `(str, bool)`. The string is the prompt and the boolean indicates whether the user text should be echoed.

A sample call would thus be: `handler('title', 'instructions', [('Password:', False)])`.

The handler should return a list or tuple of answers to the server's questions.

If the server requires multi-step authentication (which is very rare), this method will return a list of auth types permissible for the next step. Otherwise, in the normal case, an empty list is returned.

Parameters:
- **username** (*str*) – the username to authenticate as
- **handler** (*callable*) – a handler for responding to server questions
- **submethods** (*str*) – a string list of desired submethods (optional)

Returns: list of auth types permissible for the next stage of authentication (normally empty).

Raises: **BadAuthenticationType** – if public-key authentication isn't allowed by the server for this user

Raises: **AuthenticationException** – if the authentication failed

Raises: **SSHException** – if there was a network error

New in version 1.5.

auth_interactive_dumb(*username*, *handler=None*, *submethods=''*)

Autenticate to the server interactively but dumber. Just print the prompt and / or instructions to stdout and send back the response. This is good for situations where partial auth is achieved by key and then the user has to enter a 2fac token.

auth_none(*username*)

Try to authenticate to the server using no authentication at all. This will almost always fail. It may be useful for determining the list of authentication types supported by the server, by catching the **BadAuthenticationType** exception raised.

Parameters: **username** (*str*) – the username to authenticate as

Returns: list of auth types permissible for the next stage of authentication (normally empty)

Raises: **BadAuthenticationType** – if "none" authentication isn't allowed by the server for this user

Raises: **SSHException** – if the authentication failed due to a network error

New in version 1.5.

auth_password(*username*, *password*, *event=None*, *fallback=True*)

Authenticate to the server using a password. The username and password are sent over an encrypted link.

If an `event` is passed in, this method will return immediately, and the event will be triggered once authentication succeeds or fails. On success, **is_authenticated** will return `True`. On failure, you may use **get_exception** to get more detailed error information.

Since 1.1, if no event is passed, this method will block until the authentication succeeds or fails. On failure, an exception is raised. Otherwise, the method simply returns.

Since 1.5, if no event is passed and `fallback` is `True` (the default), if the server doesn't support plain password authentication but does support so-called "keyboard-interactive" mode, an attempt will be made to authenticate using this interactive mode. If it fails, the normal exception will be thrown as if the attempt had never been made. This is useful for some recent Gentoo and Debian distributions, which turn off plain password authentication in a misguided belief that interactive authentication is "more secure". (It's not.)

If the server requires multi-step authentication (which is very rare), this method will return a list of auth types permissible for the next step. Otherwise, in the normal case, an empty list is returned.

Parameters:
- **username** (*str*) – the username to authenticate as
- **password** (*basestring*) – the password to authenticate with
- **event** (*threading.Event*) – an event to trigger when the authentication attempt is complete (whether it was successful or not)
- **fallback** (*bool*) – `True` if an attempt at an automated "interactive" password auth should be made if the server doesn't support normal password auth

Returns: list of auth types permissible for the next stage of authentication (normally empty)

Raises: **BadAuthenticationType** – if password authentication isn't allowed by the server for this user (and no event was passed in)

Raises: **AuthenticationException** – if the authentication failed (and no event was passed in)

Raises: **SSHException** – if there was a network error

auth_publickey(*username*, *key*, *event=None*)

Authenticate to the server using a private key. The key is used to sign data from the server, so it must include the private part.

If an `event` is passed in, this method will return immediately, and the event will be triggered once authentication succeeds or fails. On success, **is_authenticated** will return `True`. On failure, you may use **get_exception** to get more detailed error information.

Since 1.1, if no event is passed, this method will block until the authentication succeeds or fails. On failure, an exception is raised. Otherwise, the method simply returns.

If the server requires multi-step authentication (which is very rare), this method will return a list of auth types permissible for the next step. Otherwise, in the normal case, an empty list is returned.

Parameters:
- **username** (*str*) – the username to authenticate as
- **key** (*PKey*) – the private key to authenticate with
- **event** (*threading.Event*) – an event to trigger when the authentication attempt is complete (whether it was successful or not)

Returns: list of auth types permissible for the next stage of authentication (normally empty)

Raises: **BadAuthenticationType** – if public-key authentication isn't allowed by the server for this user (and no event was passed in)

Raises: **AuthenticationException** – if the authentication failed (and no event was passed in)

Raises: **SSHException** – if there was a network error

cancel_port_forward(*address*, *port*)

Ask the server to cancel a previous port-forwarding request. No more connections to the given address & port will be forwarded across this ssh connection.

Parameters:
- **address** (*str*) – the address to stop forwarding
- **port** (*int*) – the port to stop forwarding

close()

Close this session, and any open channels that are tied to it.

connect(*hostkey=None*, *username=''*, *password=None*, *pkey=None*, *gss_host=None*, *gss_auth=False*, *gss_kex=False*, *gss_deleg_creds=True*, *gss_trust_dns=True*)

Negotiate an SSH2 session, and optionally verify the server's host key and authenticate using a password or private key. This is a shortcut for **start_client**, **get_remote_server_key**, and **Transport.auth_password** or **Transport.auth_publickey**. Use those methods if you want more control.

You can use this method immediately after creating a Transport to negotiate encryption with a server. If it fails, an exception will be thrown. On success, the method will return cleanly, and an encrypted session exists. You may immediately call **open_channel** or **open_session** to get a **Channel** object, which is used for data transfer.

> Note:
> If you fail to supply a password or private key, this method may succeed, but a subsequent **open_channel** or **open_session** call may fail because you haven't authenticated yet.

Parameters:
- **hostkey** (*PKey*) – the host key expected from the server, or `None` if you don't want to do host key verification.
- **username** (*str*) – the username to authenticate as.
- **password** (*str*) – a password to use for authentication, if you want to use password authentication; otherwise `None`.
- **pkey** (*PKey*) – a private key to use for authentication, if you want to use private key authentication; otherwise `None`.
- **gss_host** (*str*) – The target's name in the kerberos database. Default: hostname
- **gss_auth** (*bool*) – `True` if you want to use GSS-API authentication.

- **gss_kex** (*bool*) – Perform GSS-API Key Exchange and user authentication.
- **gss_deleg_creds** (*bool*) – Whether to delegate GSS-API client credentials.
- **gss_trust_dns** – Indicates whether or not the DNS is trusted to securely canonicalize the name of the host being connected to (default `True`).

Raises: **SSHException** – if the SSH2 negotiation fails, the host key supplied by the server is incorrect, or authentication fails.

Changed in version 2.3: Added the `gss_trust_dns` argument.

get_banner()

Return the banner supplied by the server upon connect. If no banner is supplied, this method returns `None`.

Returns: server supplied banner (**str**), or `None`.

New in version 1.13.

get_exception()

Return any exception that happened during the last server request. This can be used to fetch more specific error information after using calls like **start_client**. The exception (if any) is cleared after this call.

Returns: an exception, or `None` if there is no stored exception.

New in version 1.1.

get_hexdump()

Return `True` if the transport is currently logging hex dumps of protocol traffic.

Returns: `True` if hex dumps are being logged, else `False`.

New in version 1.4.

get_log_channel()

Return the channel name used for this transport's logging.

Returns: channel name as a **str**

New in version 1.2.

get_remote_server_key()

Return the host key of the server (in client mode).

> **Note:**
> Previously this call returned a tuple of `(key type, key string)`. You can get the same effect by calling **PKey.get_name** for the key type, and `str(key)` for the key string.

Raises: **SSHException** – if no session is currently active.

Returns: public key (**PKey**) of the remote server

get_security_options()

Return a **SecurityOptions** object which can be used to tweak the encryption algorithms this transport will permit (for encryption, digest/hash operations, public keys, and key exchanges) and the order of preference for them.

get_server_key()

Return the active host key, in server mode. After negotiating with the client, this method will return the negotiated host key. If only one type of host key was set with **add_server_key**, that's the only key that will ever be returned. But in cases where you have set more than one type of host key (for example, an RSA key and a DSS key), the key type will be negotiated by the client, and this method will return the key of the type agreed on. If the host key has not been negotiated yet, `None` is returned. In client mode, the behavior is undefined.

Returns: host key (**PKey**) of the type negotiated by the client, or `None`.

get_username()

Return the username this connection is authenticated for. If the session is not authenticated (or authentication failed), this method returns `None`.

Returns: username that was authenticated (a `str`), or `None`.

`getpeername`()

Return the address of the remote side of this Transport, if possible.

This is effectively a wrapper around `getpeername` on the underlying socket. If the socket-like object has no `getpeername` method, then `("unknown", 0)` is returned.

Returns: the address of the remote host, if known, as a `(str, int)` tuple.

`global_request`(*kind*, *data=None*, *wait=True*)

Make a global request to the remote host. These are normally extensions to the SSH2 protocol.

Parameters:
- **kind** (*str*) – name of the request.
- **data** (*tuple*) – an optional tuple containing additional data to attach to the request.
- **wait** (*bool*) – `True` if this method should not return until a response is received; `False` otherwise.

Returns: a `Message` containing possible additional data if the request was successful (or an empty `Message` if `wait` was `False`); `None` if the request was denied.

`is_active`()

Return true if this session is active (open).

Returns: True if the session is still active (open); False if the session is closed

`is_authenticated`()

Return true if this session is active and authenticated.

Returns: True if the session is still open and has been authenticated successfully; False if authentication failed and/or the session is closed.

static `load_server_moduli`(*filename=None*)

(optional) Load a file of prime moduli for use in doing group-exchange key negotiation in server mode. It's a rather obscure option and can be safely ignored.

In server mode, the remote client may request "group-exchange" key negotiation, which asks the server to send a random prime number that fits certain criteria. These primes are pretty difficult to compute, so they can't be generated on demand. But many systems contain a file of suitable primes (usually named something like `/etc/ssh/moduli`). If you call `load_server_moduli` and it returns `True`, then this file of primes has been loaded and we will support "group-exchange" in server mode. Otherwise server mode will just claim that it doesn't support that method of key negotiation.

Parameters: **filename** (*str*) – optional path to the moduli file, if you happen to know that it's not in a standard location.
Returns: True if a moduli file was successfully loaded; False otherwise.

> Note:
> This has no effect when used in client mode.

`open_channel`(*kind*, *dest_addr=None*, *src_addr=None*, *window_size=None*, *max_packet_size=None*, *timeout=None*)

Request a new channel to the server. `Channels` are socket-like objects used for the actual transfer of data across the session. You may only request a channel after negotiating encryption (using `connect` or `start_client`) and authenticating.

> Note:
> Modifying the the window and packet sizes might have adverse effects on the channel created. The default values are the same as in the OpenSSH code base and have been battle tested.

Parameters:
- **kind** (*str*) – the kind of channel requested (usually `"session"`, `"forwarded-tcpip"`, `"direct-tcpip"`, or `"x11"`)
- **dest_addr** (*tuple*) – the destination address (address + port tuple) of this port forwarding, if `kind` is `"forwarded-tcpip"` or `"direct-tcpip"` (ignored for other channel types)
- **src_addr** – the source address of this port forwarding, if `kind` is `"forwarded-tcpip"`, `"direct-tcpip"`, or `"x11"`
- **window_size** (*int*) – optional window size for this session.

- **max_packet_size** (*int*) – optional max packet size for this session.
- **timeout** (*float*) – optional timeout opening a channel, default 3600s (1h)

Returns: a new **Channel** on success

Raises: **SSHException** – if the request is rejected, the session ends prematurely or there is a timeout openning a channel

Changed in version 1.15: Added the `window_size` and `max_packet_size` arguments.

open_forward_agent_channel()

Request a new channel to the client, of type `"auth-agent@openssh.com"`.

This is just an alias for `open_channel('auth-agent@openssh.com')`.

Returns: a new **Channel**

Raises: **SSHException** – if the request is rejected or the session ends prematurely

open_forwarded_tcpip_channel(*src_addr*, *dest_addr*)

Request a new channel back to the client, of type `forwarded-tcpip`.

This is used after a client has requested port forwarding, for sending incoming connections back to the client.

Parameters:
- **src_addr** – originator's address
- **dest_addr** – local (server) connected address

open_session(*window_size=None*, *max_packet_size=None*, *timeout=None*)

Request a new channel to the server, of type `"session"`. This is just an alias for calling **open_channel** with an argument of `"session"`.

> **Note:**
> Modifying the the window and packet sizes might have adverse effects on the session created. The default values are the same as in the OpenSSH code base and have been battle tested.

Parameters:
- **window_size** (*int*) – optional window size for this session.
- **max_packet_size** (*int*) – optional max packet size for this session.

Returns: a new **Channel**

Raises: **SSHException** – if the request is rejected or the session ends prematurely

Changed in version 1.13.4/1.14.3/1.15.3: Added the `timeout` argument.

Changed in version 1.15: Added the `window_size` and `max_packet_size` arguments.

open_sftp_client()

Create an SFTP client channel from an open transport. On success, an SFTP session will be opened with the remote host, and a new **SFTPClient** object will be returned.

Returns: a new **SFTPClient** referring to an sftp session (channel) across this transport

open_x11_channel(*src_addr=None*)

Request a new channel to the client, of type `"x11"`. This is just an alias for `open_channel('x11', src_addr=src_addr)`.

Parameters: **src_addr** (*tuple*) – the source address (`(str, int)`) of the x11 server (port is the x11 port, ie. 6010)

Returns: a new **Channel**

Raises: **SSHException** – if the request is rejected or the session ends prematurely

renegotiate_keys()

Force this session to switch to new keys. Normally this is done automatically after the session hits a certain number of packets or bytes sent or received, but this method gives you the option of forcing new keys whenever you want. Negotiating new keys causes a pause in traffic both ways as the two sides swap keys and do computations. This method returns when the session has switched to new keys.

Raises: **SSHException** – if the key renegotiation failed (which causes the session to end)

request_port_forward(*address*, *port*, *handler=None*)

Ask the server to forward TCP connections from a listening port on the server, across this SSH session.

If a handler is given, that handler is called from a different thread whenever a forwarded connection arrives. The handler parameters are:

```
handler(
    channel,
    (origin_addr, origin_port),
    (server_addr, server_port),
)
```

where `server_addr` and `server_port` are the address and port that the server was listening on.

If no handler is set, the default behavior is to send new incoming forwarded connections into the accept queue, to be picked up via **accept**.

Parameters:
- **address** (*str*) – the address to bind when forwarding
- **port** (*int*) – the port to forward, or 0 to ask the server to allocate any port
- **handler** (*callable*) – optional handler for incoming forwarded connections, of the form `func(Channel, (str, int), (str, int))`.

Returns: the port number (**int**) allocated by the server

Raises: **SSHException** – if the server refused the TCP forward request

send_ignore(*byte_count=None*)

Send a junk packet across the encrypted link. This is sometimes used to add "noise" to a connection to confuse would-be attackers. It can also be used as a keep-alive for long lived connections traversing firewalls.

Parameters: **byte_count** (*int*) – the number of random bytes to send in the payload of the ignored packet – defaults to a random number from 10 to 41.

set_gss_host(*gss_host*, *trust_dns=True*, *gssapi_requested=True*)

Normalize/canonicalize `self.gss_host` depending on various factors.

Parameters:
- **gss_host** (*str*) – The explicitly requested GSS-oriented hostname to connect to (i.e. what the host's name is in the Kerberos database.) Defaults to `self.hostname` (which will be the 'real' target hostname and/or host portion of given socket object.)
- **trust_dns** (*bool*) – Indicates whether or not DNS is trusted; if true, DNS will be used to canonicalize the GSS hostname (which again will either be `gss_host` or the transport's default hostname.) (Defaults to True due to backwards compatibility.)
- **gssapi_requested** (*bool*) – Whether GSSAPI key exchange or authentication was even requested. If not, this is a no-op and nothing happens (and `self.gss_host` is not set.) (Defaults to True due to backwards compatibility.)

Returns: `None`.

set_hexdump(*hexdump*)

Turn on/off logging a hex dump of protocol traffic at DEBUG level in the logs. Normally you would want this off (which is the default), but if you are debugging something, it may be useful.

Parameters: **hexdump** (*bool*) – `True` to log protocol traffix (in hex) to the log; `False` otherwise.

set_keepalive(*interval*)

Turn on/off keepalive packets (default is off). If this is set, after `interval` seconds without sending any data over the connection, a "keepalive" packet will be sent (and ignored by the remote host). This can be useful to keep connections alive over a NAT, for example.

Parameters: **interval** (*int*) – seconds to wait before sending a keepalive packet (or 0 to disable keepalives).

set_log_channel(*name*)

Set the channel for this transport's logging. The default is `"paramiko.transport"` but it can be set to anything you want. (See the **logging** module for more info.) SSH Channels will log to a sub-channel of the one specified.

Parameters: **name** (*str*) – new channel name for logging

New in version 1.1.

set_subsystem_handler(*name*, *handler*, **larg*, ***kwarg*)

Set the handler class for a subsystem in server mode. If a request for this subsystem is made on an open ssh channel later, this handler will be constructed and called – see **SubsystemHandler** for more detailed documentation.

Any extra parameters (including keyword arguments) are saved and passed to the **SubsystemHandler** constructor later.

Parameters:
- **name** (*str*) – name of the subsystem.
- **handler** – subclass of **SubsystemHandler** that handles this subsystem.

start_client(*event=None*, *timeout=None*)

Negotiate a new SSH2 session as a client. This is the first step after creating a new **Transport**. A separate thread is created for protocol negotiation.

If an event is passed in, this method returns immediately. When negotiation is done (successful or not), the given `Event` will be triggered. On failure, **is_active** will return `False`.

(Since 1.4) If `event` is `None`, this method will not return until negotiation is done. On success, the method returns normally. Otherwise an SSHException is raised.

After a successful negotiation, you will usually want to authenticate, calling **auth_password** or **auth_publickey**.

> **Note:**
> **connect** is a simpler method for connecting as a client.

> **Note:**
> After calling this method (or **start_server** or **connect**), you should no longer directly read from or write to the original socket object.

Parameters:
- **event** (*threading.Event*) – an event to trigger when negotiation is complete (optional)
- **timeout** (*float*) – a timeout, in seconds, for SSH2 session negotiation (optional)

Raises: **SSHException** – if negotiation fails (and no `event` was passed in)

start_server(*event=None*, *server=None*)

Negotiate a new SSH2 session as a server. This is the first step after creating a new **Transport** and setting up your server host key(s). A separate thread is created for protocol negotiation.

If an event is passed in, this method returns immediately. When negotiation is done (successful or not), the given `Event` will be triggered. On failure, **is_active** will return `False`.

(Since 1.4) If `event` is `None`, this method will not return until negotiation is done. On success, the method returns normally. Otherwise an SSHException is raised.

After a successful negotiation, the client will need to authenticate. Override the methods **get_allowed_auths**, **check_auth_none**, **check_auth_password**, and **check_auth_publickey** in the given `server` object to control the authentication process.

After a successful authentication, the client should request to open a channel. Override **check_channel_request** in the given `server` object to allow channels to be opened.

> **Note:**
> After calling this method (or **start_client** or **connect**), you should no longer directly read from or write to the original socket object.

Parameters:
- **event** (*threading.Event*) – an event to trigger when negotiation is complete.
- **server** (*ServerInterface*) – an object used to perform authentication and create **channels**

Raises: **SSHException** – if negotiation fails (and no `event` was passed in)

use_compression(*compress=True*)

Turn on/off compression. This will only have an affect before starting the transport (ie before calling **connect**, etc). By default, compression is off since it negatively affects interactive sessions.

Parameters: **compress** (*bool*) – `True` to ask the remote client/server to compress traffic; `False` to refuse compression

New in version 1.5.2.