Channel

Abstraction for an SSH2 channel.

class **paramiko.channel.Channel**(*chanid*)

A secure tunnel across an SSH **Transport**. A Channel is meant to behave like a socket, and has an API that should be indistinguishable from the Python socket API.

Because SSH2 has a windowing kind of flow control, if you stop reading data from a Channel and its buffer fills up, the server will be unable to send you any more data until you read some of it. (This won't affect other channels on the same transport – all channels on a single transport are flow-controlled independently.) Similarly, if the server isn't reading data you send, calls to **send** may block, unless you set a timeout. This is exactly like a normal network socket, so it shouldn't be too surprising.

Instances of this class may be used as context managers.

__init__(*chanid*) ¶

Create a new channel. The channel is not associated with any particular session or **Transport** until the Transport attaches it. Normally you would only call this method from the constructor of a subclass of **Channel**.

Parameters: **chanid** (*int*) – the ID of this channel, as passed by an existing **Transport**.

__repr__()

Return a string representation of this object, for debugging.

close()

Close the channel. All future read/write operations on the channel will fail. The remote end will receive no more data (after queued data is flushed). Channels are automatically closed when their **Transport** is closed or when they are garbage collected.

exec_command(**args*, ***kwds*)

Execute a command on the server. If the server allows it, the channel will then be directly connected to the stdin, stdout, and stderr of the command being executed.

When the command finishes executing, the channel will be closed and can't be reused. You must open a new channel if you wish to execute another command.

Parameters: **command** (*str*) – a shell command to execute.
Raises: **SSHException** – if the request was rejected or the channel was closed

exit_status_ready()

Return true if the remote process has exited and returned an exit status. You may use this to poll the process status if you don't want to block in **recv_exit_status**. Note that the server may not return an exit status in some cases (like bad servers).

Returns: `True` if **recv_exit_status** will return immediately, else `False`.

New in version 1.7.3.

fileno()

Returns an OS-level file descriptor which can be used for polling, but but not for reading or writing. This is primarily to allow Python's `select` module to work.

The first time `fileno` is called on a channel, a pipe is created to simulate real OS-level file descriptor (FD) behavior. Because of this, two OS-level FDs are created, which will use up FDs faster than normal. (You won't notice this effect unless you have hundreds of channels open at the same time.)

Returns: an OS-level file descriptor (**int**)

> Warning:
> This method causes channel reads to be slightly less efficient.

get_id()

Return the **int** ID # for this channel.

The channel ID is unique across a **Transport** and usually a small number. It's also the number passed to **ServerInterface.check_channel_request** when determining whether to accept a channel request in server mode.

get_name()

Get the name of this channel that was previously set by **set_name**.

get_pty(*args, **kwds)

Request a pseudo-terminal from the server. This is usually used right after creating a client channel, to ask the server to provide some basic terminal semantics for a shell invoked with **invoke_shell**. It isn't necessary (or desirable) to call this method if you're going to execute a single command with **exec_command**.

Parameters:
- **term** (*str*) – the terminal type to emulate (for example, `'vt100'`)
- **width** (*int*) – width (in characters) of the terminal screen
- **height** (*int*) – height (in characters) of the terminal screen
- **width_pixels** (*int*) – width (in pixels) of the terminal screen
- **height_pixels** (*int*) – height (in pixels) of the terminal screen

Raises: **SSHException** – if the request was rejected or the channel was closed

get_transport()

Return the **Transport** associated with this channel.

getpeername()

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

This simply wraps **Transport.getpeername**, used to provide enough of a socket-like interface to allow asyncore to work. (asyncore likes to call `'getpeername'`.)

gettimeout()

Returns the timeout in seconds (as a float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to **setblocking** or **settimeout**.

invoke_shell(*args, **kwds)

Request an interactive shell session on this channel. If the server allows it, the channel will then be directly connected to the stdin, stdout, and stderr of the shell.

Normally you would call **get_pty** before this, in which case the shell will operate through the pty, and the channel will be connected to the stdin and stdout of the pty.

When the shell exits, the channel will be closed and can't be reused. You must open a new channel if you wish to open another shell.

Raises: **SSHException** – if the request was rejected or the channel was closed

invoke_subsystem(*args, **kwds)

Request a subsystem on the server (for example, `sftp`). If the server allows it, the channel will then be directly connected to the requested subsystem.

When the subsystem finishes, the channel will be closed and can't be reused.

Parameters: **subsystem** (*str*) – name of the subsystem being requested.
Raises: **SSHException** – if the request was rejected or the channel was closed

makefile(*params)

Return a file-like object associated with this channel. The optional `mode` and `bufsize` arguments are interpreted the same way as by the built-in `file()` function in Python.

Returns: **ChannelFile** object which can be used for Python file I/O.

makefile_stderr(*params)

Return a file-like object associated with this channel's stderr stream. Only channels using **exec_command** or **invoke_shell** without a pty will ever have data on the stderr stream.

The optional `mode` and `bufsize` arguments are interpreted the same way as by the built-in `file()` function in Python. For a client, it only makes sense to open this file for reading. For a server, it only makes sense to open this file for writing.

Returns: **ChannelFile** object which can be used for Python file I/O.

New in version 1.1.

recv(*nbytes*)

Receive data from the channel. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by nbytes. If a string of length zero is returned, the channel stream has closed.

Parameters: **nbytes** (*int*) – maximum number of bytes to read.
Returns: received data, as a str/bytes.
Raises: socket.timeout – if no data is ready before the timeout set by **settimeout**.

recv_exit_status()

Return the exit status from the process on the server. This is mostly useful for retrieving the results of an **exec_command**. If the command hasn't finished yet, this method will wait until it does, or until the channel is closed. If no exit status is provided by the server, -1 is returned.

> **Warning:**
> In some situations, receiving remote output larger than the current **Transport** or session's window_size (e.g. that set by the default_window_size kwarg for **Transport.__init__**) will cause **recv_exit_status** to hang indefinitely if it is called prior to a sufficiently large **Channel.recv** (or if there are no threads calling **Channel.recv** in the background).
>
> In these cases, ensuring that **recv_exit_status** is called *after* **Channel.recv** (or, again, using threads) can avoid the hang.

Returns: the exit code (as an **int**) of the process on the server.

New in version 1.2.

recv_ready()

Returns true if data is buffered and ready to be read from this channel. A False result does not mean that the channel has closed; it means you may need to wait before more data arrives.

Returns: True if a **recv** call on this channel would immediately return at least one byte; False otherwise.

recv_stderr(*nbytes*)

Receive data from the channel's stderr stream. Only channels using **exec_command** or **invoke_shell** without a pty will ever have data on the stderr stream. The return value is a string representing the data received. The maximum amount of data to be received at once is specified by nbytes. If a string of length zero is returned, the channel stream has closed.

Parameters: **nbytes** (*int*) – maximum number of bytes to read.
Returns: received data as a **str**
Raises: socket.timeout – if no data is ready before the timeout set by **settimeout**.

New in version 1.1.

recv_stderr_ready()

Returns true if data is buffered and ready to be read from this channel's stderr stream. Only channels using **exec_command** or **invoke_shell** without a pty will ever have data on the stderr stream.

Returns: True if a **recv_stderr** call on this channel would immediately return at least one byte; False otherwise.

New in version 1.1.

request_forward_agent(**args*, ***kwds*)

Request for a forward SSH Agent on this channel. This is only valid for an ssh-agent from OpenSSH !!!

Parameters: **handler** – a required callable handler to use for incoming SSH Agent connections
Returns: True if we are ok, else False (at that time we always return ok)
Raises: SSHException in case of channel problem.

request_x11(**args*, ***kwds*)

Request an x11 session on this channel. If the server allows it, further x11 requests can be made from the server to the client, when an x11 application is run in a shell session.

From RFC 4254:

```
It is RECOMMENDED that the 'x11 authentication cookie' that is
sent be a fake, random cookie, and that the cookie be checked and
replaced by the real cookie when a connection request is received.
```

If you omit the auth_cookie, a new secure random 128-bit value will be generated, used, and returned. You will need to use this value to verify incoming x11 requests and replace them with the actual local x11 cookie (which requires some knowledge of the x11 protocol).

If a handler is passed in, the handler is called from another thread whenever a new x11 connection arrives. The default handler queues up incoming x11 connections, which may be retrieved using **Transport.accept**. The handler's calling signature is:

```
handler(channel: Channel, (address: str, port: int))
```

Parameters:
- **screen_number** (*int*) – the x11 screen number (0, 10, etc.)
- **auth_protocol** (*str*) – the name of the X11 authentication method used; if none is given, `"MIT-MAGIC-COOKIE-1"` is used
- **auth_cookie** (*str*) – hexadecimal string containing the x11 auth cookie; if none is given, a secure random 128-bit value is generated
- **single_connection** (*bool*) – if True, only a single x11 connection will be forwarded (by default, any number of x11 connections can arrive over this session)
- **handler** – an optional callable handler to use for incoming X11 connections

Returns: the auth_cookie used

resize_pty(**args*, ***kwds*)

Resize the pseudo-terminal. This can be used to change the width and height of the terminal emulation created in a previous **get_pty** call.

Parameters:
- **width** (*int*) – new width (in characters) of the terminal screen
- **height** (*int*) – new height (in characters) of the terminal screen
- **width_pixels** (*int*) – new width (in pixels) of the terminal screen
- **height_pixels** (*int*) – new height (in pixels) of the terminal screen

Raises: **SSHException** – if the request was rejected or the channel was closed

send(*s*)

Send data to the channel. Returns the number of bytes sent, or 0 if the channel stream is closed. Applications are responsible for checking that all data has been sent: if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

Parameters: s (*str*) – data to send

Returns: number of bytes actually sent, as an **int**

Raises: socket.timeout – if no data could be sent before the timeout set by **settimeout**.

send_exit_status(*status*)

Send the exit status of an executed command to the client. (This really only makes sense in server mode.) Many clients expect to get some sort of status code back from an executed command after it completes.

Parameters: **status** (*int*) – the exit code of the process

New in version 1.2.

send_ready()

Returns true if data can be written to this channel without blocking. This means the channel is either closed (so any write attempt would return immediately) or there is at least one byte of space in the outbound buffer. If there is at least one byte of space in the outbound buffer, a **send** call will succeed immediately and return the number of bytes actually written.

Returns: `True` if a **send** call on this channel would immediately succeed or fail

send_stderr(*s*)

Send data to the channel on the "stderr" stream. This is normally only used by servers to send output from shell commands – clients won't use this. Returns the number of bytes sent, or 0 if the channel stream is closed. Applications are responsible

for checking that all data has been sent: if only some of the data was transmitted, the application needs to attempt delivery of the remaining data.

Parameters: s (*str*) – data to send.
Returns: number of bytes actually sent, as an **int**.
Raises: socket.timeout – if no data could be sent before the timeout set by **settimeout**.

New in version 1.1.

sendall(*s*)

Send data to the channel, without allowing partial results. Unlike **send**, this method continues to send data from the given string until either all data has been sent or an error occurs. Nothing is returned.

Parameters: s (*str*) – data to send.

Raises:
- socket.timeout – if sending stalled for longer than the timeout set by **settimeout**.
- socket.error – if an error occurred before the entire string was sent.

> **Note:**
> If the channel is closed while only part of the data has been sent, there is no way to determine how much data (if any) was sent. This is irritating, but identically follows Python's API.

sendall_stderr(*s*)

Send data to the channel's "stderr" stream, without allowing partial results. Unlike **send_stderr**, this method continues to send data from the given string until all data has been sent or an error occurs. Nothing is returned.

Parameters: s (*str*) – data to send to the client as "stderr" output.

Raises:
- socket.timeout – if sending stalled for longer than the timeout set by **settimeout**.
- socket.error – if an error occurred before the entire string was sent.

New in version 1.1.

set_combine_stderr(*combine*)

Set whether stderr should be combined into stdout on this channel. The default is `False`, but in some cases it may be convenient to have both streams combined.

If this is `False`, and **exec_command** is called (or `invoke_shell` with no pty), output to stderr will not show up through the **recv** and **recv_ready** calls. You will have to use **recv_stderr** and **recv_stderr_ready** to get stderr output.

If this is `True`, data will never show up via **recv_stderr** or **recv_stderr_ready**.

Parameters: **combine** (*bool*) – `True` if stderr output should be combined into stdout on this channel.
Returns: the previous setting (a **bool**).

New in version 1.1.

set_environment_variable(**args*, ***kwds*)

Set the value of an environment variable.

> **Warning:**
> The server may reject this request depending on its `AcceptEnv` setting; such rejections will fail silently (which is common client practice for this particular request type). Make sure you understand your server's configuration before using!

Parameters:
- **name** (*str*) – name of the environment variable
- **value** (*str*) – value of the environment variable

Raises: **SSHException** – if the request was rejected or the channel was closed

set_name(*name*)

Set a name for this channel. Currently it's only used to set the name of the channel in logfile entries. The name can be fetched with the **get_name** method.

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

setblocking(*blocking*)

Set blocking or non-blocking mode of the channel: if `blocking` is 0, the channel is set to non-blocking mode; otherwise it's set to blocking mode. Initially all channels are in blocking mode.

In non-blocking mode, if a **recv** call doesn't find any data, or if a **send** call can't immediately dispose of the data, an error exception is raised. In blocking mode, the calls block until they can proceed. An EOF condition is considered "immediate data" for **recv**, so if the channel is closed in the read direction, it will never block.

`chan.setblocking(0)` is equivalent to `chan.settimeout(0)`; `chan.setblocking(1)` is equivalent to `chan.settimeout(None)`.

Parameters: **blocking** (*int*) – 0 to set non-blocking mode; non-0 to set blocking mode.

settimeout(*timeout*)

Set a timeout on blocking read/write operations. The `timeout` argument can be a nonnegative float expressing seconds, or `None`. If a float is given, subsequent channel read/write operations will raise a timeout exception if the timeout period value has elapsed before the operation has completed. Setting a timeout of `None` disables timeouts on socket operations.

`chan.settimeout(0.0)` is equivalent to `chan.setblocking(0)`; `chan.settimeout(None)` is equivalent to `chan.setblocking(1)`.

Parameters: **timeout** (*float*) – seconds to wait for a pending read/write operation before raising `socket.timeout`, or `None` for no timeout.

shutdown(*how*)

Shut down one or both halves of the connection. If `how` is 0, further receives are disallowed. If `how` is 1, further sends are disallowed. If `how` is 2, further sends and receives are disallowed. This closes the stream in one or both directions.

Parameters: **how** (*int*) –
0 (stop receiving), 1 (stop sending), or 2 (stop receiving and sending).

shutdown_read()

Shutdown the receiving side of this socket, closing the stream in the incoming direction. After this call, future reads on this channel will fail instantly. This is a convenience method, equivalent to `shutdown(0)`, for people who don't make it a habit to memorize unix constants from the 1970s.

New in version 1.2.

shutdown_write()

Shutdown the sending side of this socket, closing the stream in the outgoing direction. After this call, future writes on this channel will fail instantly. This is a convenience method, equivalent to `shutdown(1)`, for people who don't make it a habit to memorize unix constants from the 1970s.

New in version 1.2.

update_environment(**args*, ***kwds*)

Updates this channel's remote shell environment.

> **Note:**
> This operation is additive - i.e. the current environment is not reset before the given environment variables are set.

> **Warning:**
> Servers may silently reject some environment variables; see the warning in **set_environment_variable** for details.

Parameters: **environment** (*dict*) – a dictionary containing the name and respective values to set
Raises: **SSHException** – if any of the environment variables was rejected by the server or the channel was closed

class **paramiko.channel.ChannelFile**(*channel*, *mode='r'*, *bufsize=-1*)

A file-like wrapper around **Channel**. A ChannelFile is created by calling **Channel.makefile**.

> **Warning:**
> To correctly emulate the file object created from a socket's **`makefile`** method, a **`Channel`** and its **`ChannelFile`** should be able to be closed or garbage-collected independently. Currently, closing the **`ChannelFile`** does nothing but flush the buffer.

`__repr__`()

Returns a string representation of this object, for debugging.

`paramiko.channel.`**`open_only`**(*func*)

Decorator for **`Channel`** methods which performs an openness check.

Raises: **`SSHException`** – If the wrapped method is called on an unopened **`Channel`**.