hyper Documentation

Release 0.5.0

Cory Benfield

October 29, 2015

Contents

Release v0.5.0.

HTTP is changing under our feet. HTTP/1.1, our old friend, is being supplemented by the brand new HTTP/2 standard. HTTP/2 provides many benefits: improved speed, lower bandwidth usage, better connection management, and more.

`hyper` provides these benefits to your Python code. How? Like this:

```
from hyper import HTTPConnection

conn = HTTPConnection('http2bin.org:443')
conn.request('GET', '/get')
resp = conn.get_response()

print(resp.read())
```

Simple. `hyper` is written in 100% pure Python, which means no C extensions. For recent versions of Python (3.4 and onward, and 2.7.9 and onward) it's entirely self-contained with no external dependencies.

`hyper` supports Python 3.4 and Python 2.7.9, and can speak HTTP/2 and HTTP/1.1.

CHAPTER 1

Caveat Emptor!

Please be warned: `hyper` is in a very early alpha. You *will* encounter bugs when using it. In addition, there are very many rough edges. With that said, please try it out in your applications: I need your feedback to fix the bugs and file down the rough edges.

CHAPTER 2

Get Stuck In

The quickstart documentation will help get you going with `hyper`.

2.1 Quickstart Guide

First, congratulations on picking `hyper` for your HTTP needs. `hyper` is the premier (and, as far as we're aware, the only) Python HTTP/2 library, as well as a very serviceable HTTP/1.1 library.

In this section, we'll walk you through using `hyper`.

2.1.1 Installing hyper

To begin, you will need to install `hyper`. This can be done like so:

```
$ pip install hyper
```

If `pip` is not available to you, you can try:

```
$ easy_install hyper
```

If that fails, download the library from its GitHub page and install it using:

```
$ python setup.py install
```

Installation Requirements

The HTTP/2 specification requires very modern TLS support from any compliant implementation. When using Python 3.4 and later this is automatically provided by the standard library. For earlier releases of Python, we use PyOpenSSL to provide the TLS support we need.

Unfortunately, this is not always totally trivial. You will need to build PyOpenSSL against a version of OpenSSL that is at least 1.0.1, and to do that you'll actually need to obtain that version of OpenSSL.

To install against the relevant version of OpenSSL for your system, follow the instructions from the cryptography project, replacing references to `cryptography` with `hyper`.

2.1.2 Making Your First HTTP Request

With `hyper` installed, you can start making HTTP/2 requests. For the rest of these examples, we'll use http2bin.org, a HTTP/1.1 and HTTP/2 testing service.

Begin by getting the homepage:

```
>>> from hyper import HTTPConnection
>>> c = HTTPConnection('http2bin.org')
>>> c.request('GET', '/')
1
>>> resp = c.get_response()
```

Used in this way, `hyper` behaves exactly like `http.client`. You can make sequential requests using the exact same API you're accustomed to. The only difference is that *`HTTPConnection.request()`* may return a value, unlike the equivalent `http.client` function. If present, the return value is the HTTP/2 *stream identifier*. If you're planning to use `hyper` in this very simple way, you can choose to ignore it, but it's potentially useful. We'll come back to it.

Once you've got the data, things diverge a little bit:

```
>>> resp.headers['content-type']
[b'text/html; charset=utf-8']
>>> resp.headers
HTTPHeaderMap([(b'server', b'h2o/1.0.2-alpha1')...
>>> resp.status
200
```

If http2bin had compressed the response body then `hyper` would automatically decompress that body for you, no input required. This means you can always get the body by simply reading it:

```
>>> body = resp.read()
b'<!DOCTYPE html>\n<!--[if IE 8]><html clas ....
```

That's all it takes.

2.1.3 Streams

In HTTP/2, connections are divided into multiple streams. Each stream carries a single request-response pair. You may start multiple requests before reading the response from any of them, and switch between them using their stream IDs.

For example:

```
>>> from hyper import HTTPConnection
>>> c = HTTPConnection('http2bin.org')
>>> first = c.request('GET', '/get', headers={'key': 'value'})
>>> second = c.request('POST', '/post', body=b'hello')
>>> third = c.request('GET', '/ip')
>>> second_response = c.get_response(second)
>>> first_response = c.get_response(first)
>>> third_response = c.get_response(third)
```

`hyper` will ensure that each response is matched to the correct request.

2.1.4 Abstraction

When you use the *HTTPConnection* object, you don't have to know in advance whether your service supports HTTP/2 or not. If it doesn't, hyper will transparently fall back to HTTP/1.1.

You can tell the difference: if *request* returns a stream ID, then the connection is using HTTP/2: if it returns None, then HTTP/1.1 is being used.

Generally, though, you don't need to care.

2.1.5 Requests Integration

Do you like requests? Of course you do, everyone does! It's a shame that requests doesn't support HTTP/2 though. To rectify that oversight, hyper provides a transport adapter that can be plugged directly into Requests, giving it instant HTTP/2 support.

Using hyper with requests is super simple:

```
>>> import requests
>>> from hyper.contrib import HTTP20Adapter
>>> s = requests.Session()
>>> s.mount('https://http2bin.org', HTTP20Adapter())
>>> r = s.get('https://http2bin.org/get')
>>> print(r.status_code)
200
```

This transport adapter is subject to all of the limitations that apply to hyper, and provides all of the goodness of requests.

2.1.6 HTTPie Integration

HTTPie is a popular tool for making HTTP requests from the command line, as an alternative to the ever-popular cURL. Collaboration between the hyper authors and the HTTPie authors allows HTTPie to support making HTTP/2 requests.

To add this support, follow the instructions in the GitHub repository.

2.1.7 hyper CLI

For testing purposes, hyper provides a command-line tool that can make HTTP/2 requests directly from the CLI. This is useful for debugging purposes, and to avoid having to use the Python interactive interpreter to execute basic queries.

For more information, see the CLI section.

CHAPTER 3

Advanced Documentation

More advanced topics are covered here.

3.1 Advanced Usage

This section of the documentation covers more advanced use-cases for `hyper`.

3.1.1 Responses as Context Managers

If you're concerned about having too many TCP sockets open at any one time, you may want to keep your connections alive only as long as you know you'll need them. In HTTP/2 this is generally not something you should do unless you're very confident you won't need the connection again anytime soon. However, if you decide you want to avoid keeping the connection open, you can use the *`HTTPConnection`* as a context manager:

```
with HTTPConnection('http2bin.org') as conn:
    conn.request('GET', '/get')
    data = conn.get_response().read()

analyse(data)
```

You may not use any *`HTTP20Response`* or *`HTTP11Response`* objects obtained from a connection after that connection is closed. Interacting with these objects when a connection has been closed is considered undefined behaviour.

3.1.2 Chunked Responses

Plenty of APIs return chunked data, and it's often useful to iterate directly over the chunked data. `hyper` lets you iterate over each data frame of a HTTP/2 response, and each chunk of a HTTP/1.1 response delivered with `Transfer-Encoding: chunked`:

```
for chunk in response.read_chunked():
    do_something_with_chunk(chunk)
```

There are some important caveats with this iteration: mostly, it's not guaranteed that each chunk will be non-empty. In HTTP/2, it's entirely legal to send zero-length data frames, and this API will pass those through unchanged. Additionally, by default this method will decompress a response that has a compressed `Content-Encoding`: if you do that, each element of the iterator will no longer be a single chunk, but will instead be whatever the decompressor returns for that chunk.

If that's problematic, you can set the `decode_content` parameter to `False` and, if necessary, handle the decompression yourself:

```
for compressed_chunk in response.read_chunked(decode_content=False):
    decompress(compressed_chunk)
```

Very easy!

3.1.3 Multithreading

Currently, `hyper`'s *`HTTPConnection`* class is **not** thread-safe. Thread-safety is planned for `hyper`'s core objects, but in this early alpha it is not a high priority.

To use `hyper` in a multithreaded context the recommended thing to do is to place each connection in its own thread. Each thread should then have a request queue and a response queue, and the thread should be able to spin over both, sending requests and returning responses. The stream identifiers provided by `hyper` can be used to match the two together.

3.1.4 SSL/TLS Certificate Verification

By default, all HTTP/2 connections are made over TLS, and `hyper` bundles certificate authorities that it uses to verify the offered TLS certificates. Currently certificate verification cannot be disabled.

3.1.5 Streaming Uploads

Just like the ever-popular `requests` module, `hyper` allows you to perform a 'streaming' upload by providing a file-like object to the 'data' parameter. This will cause `hyper` to read the data in 1kB at a time and send it to the remote server. You *must* set an accurate Content-Length header when you do this, as `hyper` won't set it for you.

3.1.6 Content Decompression

In HTTP/2 it's mandatory that user-agents support receiving responses that have their bodies compressed. As demonstrated in the quickstart guide, `hyper` transparently implements this decompression, meaning that responses are automatically decompressed for you. If you don't want this to happen, you can turn it off by passing the `decode_content` parameter to *`read()`*, like this:

```
>>> resp.read(decode_content=False)
b'\xc9...'
```

3.1.7 Flow Control & Window Managers

HTTP/2 provides a facility for performing 'flow control', enabling both ends of a HTTP/2 connection to influence the rate at which data is received. When used correctly flow control can be a powerful tool for maximising the efficiency of a connection. However, when used poorly, flow control leads to severe inefficiency and can adversely affect the throughput of the connection.

By default `hyper` does its best to manage the flow control window for you, trying to avoid severe inefficiencies. In general, though, the user has a much better idea of how to manage the flow control window than `hyper` will: you know your use case better than `hyper` possibly can.

For that reason, `hyper` provides a facility for using pluggable *window managers*. A *window manager* is an object that is in control of resizing the flow control window. This object gets informed about every frame received on the

connection, and can make decisions about when to increase the size of the receive window. This object can take advantage of knowledge from layers above `hyper`, in the user's code, as well as knowledge from `hyper`'s layer.

To implement one of these objects, you will want to subclass the *`BaseFlowControlManager`* class and implement the *`increase_window_size()`* method. As a simple example, we can implement a very stupid flow control manager that always resizes the window in response to incoming data like this:

```
class StupidFlowControlManager(BaseFlowControlManager):
    def increase_window_size(self, frame_size):
        return frame_size
```

The *class* can then be plugged straight into a connection object:

```
HTTP20Connection('http2bin.org', window_manager=StupidFlowControlManager)
```

Note that we don't plug an instance of the class in, we plug the class itself in. We do this because the connection object will spawn instances of the class in order to manage the flow control windows of streams in addition to managing the window of the connection itself.

3.1.8 Server Push

HTTP/2 provides servers with the ability to "push" additional resources to clients in response to a request, as if the client had requested the resources themselves. When minimizing the number of round trips is more critical than maximizing bandwidth usage, this can be a significant performance improvement.

Servers may declare their intention to push a given resource by sending the headers and other metadata of a request that would return that resource - this is referred to as a "push promise". They may do this before sending the response headers for the original request, after, or in the middle of sending the response body.

In order to receive pushed resources, the *`HTTPConnection`* object must be constructed with `enable_push=True`.

You may retrieve the push promises that the server has sent *so far* by calling *`get_pushes()`*, which returns a generator that yields *`HTTP20Push`* objects. Note that this method is not idempotent; promises returned in one call will not be returned in subsequent calls. If `capture_all=False` is passed (the default), the generator will yield all buffered push promises without blocking. However, if `capture_all=True` is passed, the generator will first yield all buffered push promises, then yield additional ones as they arrive, and terminate when the original stream closes. Using this parameter is only recommended when it is known that all pushed streams, or a specific one, are of higher priority than the original response, or when also processing the original response in a separate thread (N.B. do not do this; `hyper` is not yet thread-safe):

```
conn.request('GET', '/')
response = conn.get_response()
for push in conn.get_pushes(): # all pushes promised before response headers
    print(push.path)
conn.read()
for push in conn.get_pushes(): # all other pushes
    print(push.path)
```

To cancel an in-progress pushed stream (for example, if the user already has the given path in cache), call *`HTTP20Push.cancel()`*.

`hyper` does not currently verify that pushed resources comply with the Same-Origin Policy, so users must take care that they do not treat pushed resources as authoritative without performing this check themselves (since the server push mechanism is only an optimization, and clients are free to issue requests for any pushed resources manually, there is little downside to simply ignoring suspicious ones).

3.1.9 Nghttp2

By default `hyper` uses its built-in pure-Python HPACK encoder and decoder. These are reasonably efficient, and suitable for most use cases. However, they do not produce the best compression ratio possible, and because they're written in pure-Python they incur a cost in memory usage above what is strictly necessary.

nghttp2 is a HTTP/2 library written in C that includes a HPACK encoder and decoder. `nghttp2`'s encoder produces extremely compressed output, and because it is written in C it is also fast and memory efficient. For this reason, performance conscious users may prefer to use `nghttp2`'s HPACK implementation instead of `hyper`'s.

You can do this very easily. If `nghttp2`'s Python bindings are installed, `hyper` will transparently switch to using `nghttp2`'s HPACK implementation instead of its own. No configuration is required.

Instructions for installing `nghttp2` are available here.

3.2 Hyper Command Line Interface

For testing purposes, `hyper` provides a command-line tool that can make HTTP/2 requests directly from the CLI. This is useful for debugging purposes, and to avoid having to use the Python interactive interpreter to execute basic queries.

The usage is:

```
hyper [-h] [--version] [--debug] [METHOD] URL [REQUEST_ITEM [REQUEST_ITEM ...]]
```

For example:

```
$ hyper GET https://http2bin.org/get
{'args': {},
 'headers': {'Connection': 'close', 'Host': 'http2bin.org', 'Via': '2.0 nghttpx'},
 'origin': '81.129.184.72',
 'url': 'https://http2bin.org/get'}
```

This allows making basic queries to confirm that `hyper` is functioning correctly, or to perform very basic interop testing with other services.

3.2.1 Sending Data

The `hyper` tool has a limited ability to send certain kinds of data. You can add extra headers by passing them as colon-separated data:

```
$ hyper GET https://http2bin.org/get User-Agent:hyper/0.2.0 X-Totally-Real-Header:someval
{'args': {},
 'headers': {'Connection': 'close',
             'Host': 'http2bin.org',
             'User-Agent': 'hyper/0.2.0',
             'Via': '2.0 nghttpx',
             'X-Totally-Real-Header': 'someval'},
 'origin': '81.129.184.72',
 'url': 'https://http2bin.org/get'}
```

You can add query-string parameters:

```
$ hyper GET https://http2bin.org/get search==hyper
{'args': {'search': 'hyper'},
 'headers': {'Connection': 'close', 'Host': 'http2bin.org', 'Via': '2.0 nghttpx'},
```

```
 'origin': '81.129.184.72',
 'url': 'https://http2bin.org/get?search=hyper'}
```

And you can upload JSON objects:

```
$ hyper POST https://http2bin.org/post name=Hyper language=Python description='CLI HTTP client'
{'args': {},
 'data': '{"name": "Hyper", "description": "CLI HTTP client", "language": '
         '"Python"}',
 'files': {},
 'form': {},
 'headers': {'Connection': 'close',
             'Content-Length': '73',
             'Content-Type': 'application/json; charset=utf-8',
             'Host': 'http2bin.org',
             'Via': '2.0 nghttpx'},
 'json': {'description': 'CLI HTTP client',
          'language': 'Python',
          'name': 'Hyper'},
 'origin': '81.129.184.72',
 'url': 'https://http2bin.org/post'}
```

3.2.2 Debugging and Detail

For more detail, passing the `--debug` flag will enable `hyper`'s DEBUG-level logging. This provides a lot of low-level detail about exactly what `hyper` is doing, including sent and received frames and HPACK state.

3.2.3 Notes

The `hyper` command-line tool is not intended to be a fully functional HTTP CLI tool: for that, we recommend using HTTPie, which uses `hyper` for its HTTP/2 support.

CHAPTER 4

Contributing

Want to contribute? Awesome! This guide goes into detail about how to contribute, and provides guidelines for project contributions.

4.1 Contributor's Guide

If you're reading this you're probably interested in contributing to `hyper`. First, I'd like to say: thankyou! Projects like this one live-and-die based on the support they receive from others, and the fact that you're even *considering* supporting `hyper` is incredibly generous of you.

This document lays out guidelines and advice for contributing to `hyper`. If you're thinking of contributing, start by reading this thoroughly and getting a feel for how contributing to the project works. If you've still got questions after reading this, you should go ahead and contact the author: he'll be happy to help.

The guide is split into sections based on the type of contribution you're thinking of making, with a section that covers general guidelines for all contributors.

4.1.1 All Contributions

Be Cordial Or Be On Your Way

`hyper` has one very important guideline governing all forms of contribution, including things like reporting bugs or requesting features. The guideline is be cordial or be on your way. **All contributions are welcome**, but they come with an implicit social contract: everyone must be treated with respect.

This can be a difficult area to judge, so the maintainer will enforce the following policy. If any contributor acts rudely or aggressively towards any other contributor, **regardless of whether they perceive themselves to be acting in retaliation for an earlier breach of this guideline**, they will be subject to the following steps:

1. They must apologise. This apology must be genuine in nature: "I'm sorry you were offended" is not sufficient. The judgement of 'genuine' is at the discretion of the maintainer.
2. If the apology is not offered, any outstanding and future contributions from the violating contributor will be rejected immediately.

Everyone involved in the `hyper` project, the maintainer included, is bound by this policy. Failing to abide by it leads to the offender being kicked off the project.

Get Early Feedback

If you are contributing, do not feel the need to sit on your contribution until it is perfectly polished and complete. It helps everyone involved for you to seek feedback as early as you possibly can. Submitting an early, unfinished version of your contribution for feedback in no way prejudices your chances of getting that contribution accepted, and can save you from putting a lot of work into a contribution that is not suitable for the project.

Contribution Suitability

The project maintainer has the last word on whether or not a contribution is suitable for `hyper`. All contributions will be considered, but from time to time contributions will be rejected because they do not suit the project.

If your contribution is rejected, don't despair! So long as you followed these guidelines, you'll have a much better chance of getting your next contribution accepted.

4.1.2 Code Contributions

Steps

When contributing code, you'll want to follow this checklist:

1. Fork the repository on GitHub.
2. Run the tests to confirm they all pass on your system. If they don't, you'll need to investigate why they fail. If you're unable to diagnose this yourself, raise it as a bug report by following the guidelines in this document: *Bug Reports*.
3. Write tests that demonstrate your bug or feature. Ensure that they fail.
4. Make your change.
5. Run the entire test suite again, confirming that all tests pass *including the ones you just added*.
6. Send a GitHub Pull Request to the main repository's `development` branch. GitHub Pull Requests are the expected method of code collaboration on this project. If you object to the GitHub workflow, you may mail a patch to the maintainer.

The following sub-sections go into more detail on some of the points above.

Tests & Code Coverage

`hyper` has a substantial suite of tests, both unit tests and integration tests, and has 100% code coverage. Whenever you contribute, you must write tests that exercise your contributed code, and you must not regress the code coverage.

To run the tests, you need to install tox. Once you have, you can run the tests against all supported platforms by simply executing `tox`.

If you're having trouble running the tests, please consider raising a bug report using the guidelines in the *Bug Reports* section.

If you've done this but want to get contributing right away, you can take advantage of the fact that `hyper` uses a continuous integration system. This will automatically run the tests against any pull request raised against the main `hyper` repository. The continuous integration system treats a regression in code coverage as a failure of the test suite.

Before a contribution is merged it must have a green run through the CI system.

Code Review

Contributions will not be merged until they've been code reviewed. You should implement any code review feedback unless you strongly object to it. In the event that you object to the code review feedback, you should make your case clearly and calmly. If, after doing so, the feedback is judged to still apply, you must either apply the feedback or withdraw your contribution.

New Contributors

If you are new or relatively new to Open Source, welcome! `hyper` aims to be a gentle introduction to the world of Open Source. If you're concerned about how best to contribute, please consider mailing the maintainer and asking for help.

Please also check the *Get Early Feedback* section.

4.1.3 Documentation Contributions

Documentation improvements are always welcome! The documentation files live in the `docs/` directory of the codebase. They're written in reStructuredText, and use Sphinx to generate the full suite of documentation.

When contributing documentation, please attempt to follow the style of the documentation files. This means a soft-limit of 79 characters wide in your text files and a semi-formal prose style.

4.1.4 Bug Reports

Bug reports are hugely important! Before you raise one, though, please check through the GitHub issues, **both open and closed**, to confirm that the bug hasn't been reported before. Duplicate bug reports are a huge drain on the time of other contributors, and should be avoided as much as possible.

4.1.5 Feature Requests

Feature requests are always welcome, but please note that all the general guidelines for contribution apply. Also note that the importance of a feature request *without* an associated Pull Request is always lower than the importance of one *with* an associated Pull Request: code is more valuable than ideas.

CHAPTER 5

Frequently Asked Questions

Got a question? I might have answered it already! Take a look.

5.1 Frequently Asked Questions

`hyper` is a project that's under active development, and is in early alpha. As a result, there are plenty of rough edges and bugs. This section of the documentation attempts to address some of your likely questions.

If you find there is no answer to your question in this list, please send me an email. My email address can be found on my GitHub profile page.

5.1.1 What version of the HTTP/2 specification does `hyper` support?

`hyper` supports the final version of the HTTP/2 draft specification. It also supports versions 14, 15, and 16 of the specification. It supports the final version of the HPACK draft specification.

5.1.2 Does `hyper` support HTTP/2 flow control?

It should! If you find it doesn't, that's a bug: please report it on GitHub.

5.1.3 Does `hyper` support Server Push?

Yes! See *Server Push*.

5.1.4 I hit a bug! What should I do?

Please tell me about it using the GitHub page for the project, here, by filing an issue. There will definitely be bugs as `hyper` is very new, and reporting them is the fastest way to get them fixed.

When you report them, please follow the contribution guidelines in the README. It'll make it a lot easier for me to fix the problem.

5.1.5 Updates

Further questions will be added here over time. Please check back regularly.

CHAPTER 6

API Documentation

The hyper API is documented in these pages.

6.1 Interface

This section of the documentation covers the interface portions of hyper.

6.1.1 Primary HTTP Interface

class hyper.**HTTPConnection**(*host*, *port=None*, *secure=None*, *window_manager=None*, *enable_push=False*, *ssl_context=None*, *proxy_host=None*, *proxy_port=None*, ***kwargs*)

An object representing a single HTTP connection to a server.

This object behaves similarly to the Python standard library's HTTPConnection object, with a few critical differences.

Most of the standard library's arguments to the constructor are not supported by hyper. Most optional parameters apply to *either* HTTP/1.1 or HTTP/2.

Parameters

- **host** – The host to connect to. This may be an IP address or a hostname, and optionally may include a port: for example, 'http2bin.org', 'http2bin.org:443' or '127.0.0.1'.
- **port** – (optional) The port to connect to. If not provided and one also isn't provided in the host parameter, defaults to 443.
- **secure** – (optional) Whether the request should use TLS. Defaults to False for most requests, but to True for any request issued to port 443.
- **window_manager** – (optional) The class to use to manage flow control windows. This needs to be a subclass of the *BaseFlowControlManager*. If not provided, *FlowControlManager* will be used.
- **enable_push** – (optional) Whether the server is allowed to push resources to the client (see *get_pushes()*).
- **ssl_context** – (optional) A class with custom certificate settings. If not provided then hyper's default SSLContext is used instead.

- **proxy_host** – (optional) The proxy to connect to. This can be an IP address or a host name and may include a port.
- **proxy_port** – (optional) The proxy port to connect to. If not provided and one also isn't provided in the `proxy` parameter, defaults to 8080.

get_response(**args*, ***kwargs*)
Returns a response object.

request(*method*, *url*, *body=None*, *headers={}*)
This will send a request to the server using the HTTP request method `method` and the selector `url`. If the `body` argument is present, it should be string or bytes object of data to send after the headers are finished. Strings are encoded as UTF-8. To use other encodings, pass a bytes object. The Content-Length header is set to the length of the body field.

Parameters

- **method** – The request method, e.g. `'GET'`.
- **url** – The URL to contact, e.g. `'/path/segment'`.
- **body** – (optional) The request body to send. Must be a bytestring or a file-like object.
- **headers** – (optional) The headers to send on the request.

Returns A stream ID for the request, or `None` if the request is made over HTTP/1.1.

6.1.2 HTTP/2

class `hyper.`**HTTP20Connection**(*host*, *port=None*, *secure=None*, *window_manager=None*, *enable_push=False*, *ssl_context=None*, *proxy_host=None*, *proxy_port=None*, ***kwargs*)
An object representing a single HTTP/2 connection to a server.

This object behaves similarly to the Python standard library's `HTTPConnection` object, with a few critical differences.

Most of the standard library's arguments to the constructor are irrelevant for HTTP/2 or not supported by hyper.

Parameters

- **host** – The host to connect to. This may be an IP address or a hostname, and optionally may include a port: for example, `'http2bin.org'`, `'http2bin.org:443'` or `'127.0.0.1'`.
- **port** – (optional) The port to connect to. If not provided and one also isn't provided in the `host` parameter, defaults to 443.
- **secure** – (optional) Whether the request should use TLS. Defaults to `False` for most requests, but to `True` for any request issued to port 443.
- **window_manager** – (optional) The class to use to manage flow control windows. This needs to be a subclass of the *BaseFlowControlManager*. If not provided, *FlowControlManager* will be used.
- **enable_push** – (optional) Whether the server is allowed to push resources to the client (see *get_pushes()*).
- **ssl_context** – (optional) A class with custom certificate settings. If not provided then hyper's default `SSLContext` is used instead.
- **proxy_host** – (optional) The proxy to connect to. This can be an IP address or a host name and may include a port.

- **proxy_port** – (optional) The proxy port to connect to. If not provided and one also isn't provided in the `proxy` parameter, defaults to 8080.

close(*error_code=None*)
Close the connection to the server.

Parameters **error_code** – (optional) The error code to reset all streams with.

Returns Nothing.

connect()
Connect to the server specified when the object was created. This is a no-op if we're already connected.

Returns Nothing.

endheaders(*message_body=None*, *final=False*, *stream_id=None*)
Sends the prepared headers to the server. If the `message_body` argument is provided it will also be sent to the server as the body of the request, and the stream will immediately be closed. If the `final` argument is set to True, the stream will also immediately be closed: otherwise, the stream will be left open and subsequent calls to `send()` will be required.

Parameters

- **message_body** – (optional) The body to send. May not be provided assuming that `send()` will be called.
- **final** – (optional) If the `message_body` parameter is provided, should be set to `True` if no further data will be provided via calls to *send()*.
- **stream_id** – (optional) The stream ID of the request to finish sending the headers on.

Returns Nothing.

get_pushes(*stream_id=None*, *capture_all=False*)
Returns a generator that yields push promises from the server. **Note that this method is not idempotent**: promises returned in one call will not be returned in subsequent calls. Iterating through generators returned by multiple calls to this method simultaneously results in undefined behavior.

Parameters

- **stream_id** – (optional) The stream ID of the request for which to get push promises.
- **capture_all** – (optional) If `False`, the generator will yield all buffered push promises without blocking. If `True`, the generator will first yield all buffered push promises, then yield additional ones as they arrive, and terminate when the original stream closes.

Returns A generator of *HTTP20Push* objects corresponding to the streams pushed by the server.

get_response(*stream_id=None*)
Should be called after a request is sent to get a response from the server. If sending multiple parallel requests, pass the stream ID of the request whose response you want. Returns a *HTTP20Response* instance. If you pass no `stream_id`, you will receive the oldest *HTTPResponse* still outstanding.

Parameters **stream_id** – (optional) The stream ID of the request for which to get a response.

Returns A *HTTP20Response* object.

network_buffer_size = None
The size of the in-memory buffer used to store data from the network. This is used as a performance optimisation. Increase buffer size to improve performance: decrease it to conserve memory. Defaults to 64kB.

putheader(*header*, *argument*, *stream_id=None*)
Sends an HTTP header to the server, with name `header` and value `argument`.

Unlike the `httplib` version of this function, this version does not actually send anything when called. Instead, it queues the headers up to be sent when you call *endheaders()*.

This method ensures that headers conform to the HTTP/2 specification. In particular, it strips out the `Connection` header, as that header is no longer valid in HTTP/2. This is to make it easy to write code that runs correctly in both HTTP/1.1 and HTTP/2.

Parameters

- **header** – The name of the header.
- **argument** – The value of the header.
- **stream_id** – (optional) The stream ID of the request to add the header to.

Returns Nothing.

putrequest(*method*, *selector*, ***kwargs*)
This should be the first call for sending a given HTTP request to a server. It returns a stream ID for the given connection that should be passed to all subsequent request building calls.

Parameters

- **method** – The request method, e.g. `'GET'`.
- **selector** – The path selector.

Returns A stream ID for the request.

receive_frame(*frame*)
Handles receiving frames intended for the stream.

request(*method*, *url*, *body=None*, *headers={}*)
This will send a request to the server using the HTTP request method `method` and the selector `url`. If the `body` argument is present, it should be string or bytes object of data to send after the headers are finished. Strings are encoded as UTF-8. To use other encodings, pass a bytes object. The Content-Length header is set to the length of the body field.

Parameters

- **method** – The request method, e.g. `'GET'`.
- **url** – The URL to contact, e.g. `'/path/segment'`.
- **body** – (optional) The request body to send. Must be a bytestring or a file-like object.
- **headers** – (optional) The headers to send on the request.

Returns A stream ID for the request.

send(*data*, *final=False*, *stream_id=None*)
Sends some data to the server. This data will be sent immediately (excluding the normal HTTP/2 flow control rules). If this is the last data that will be sent as part of this request, the `final` argument should be set to `True`. This will cause the stream to be closed.

Parameters

- **data** – The data to send.
- **final** – (optional) Whether this is the last bit of data to be sent on this request.
- **stream_id** – (optional) The stream ID of the request to send the data on.

Returns Nothing.

class hyper.**HTTP20Response**(*headers*, *stream*)

An HTTP20Response wraps the HTTP/2 response from the server. It provides access to the response headers and the entity body. The response is an iterable object and can be used in a with statement (though due to the persistent connections used in HTTP/2 this has no effect, and is done soley for compatibility).

close()

Close the response. In effect this closes the backing HTTP/2 stream.

Returns Nothing.

fileno()

Return the fileno of the underlying socket. This function is currently not implemented.

headers = None

The response headers. These are determined upon creation, assigned once, and never assigned again.

read(*amt=None*, *decode_content=True*)

Reads the response body, or up to the next amt bytes.

Parameters

- **amt** – (optional) The amount of data to read. If not provided, all the data will be read from the response.
- **decode_content** – (optional) If True, will transparently decode the response data.

Returns The read data. Note that if decode_content is set to True, the actual amount of data returned may be different to the amount requested.

read_chunked(*decode_content=True*)

Reads chunked transfer encoded bodies. This method returns a generator: each iteration of which yields one data frame *unless* the frames contain compressed data and decode_content is True, in which case it yields whatever the decompressor provides for each chunk.

Warning: This may yield the empty string, without that being the end of the body!

reason = None

The reason phrase returned by the server. This is not used in HTTP/2, and so is always the empty string.

status = None

The status code returned by the server.

trailers

Trailers on the HTTP message, if any.

Warning: Note that this property requires that the stream is totally exhausted. This means that, if you have not completely read from the stream, all stream data will be read into memory.

class hyper.**HTTP20Push**(*request_headers*, *stream*)

Represents a request-response pair sent by the server through the server push mechanism.

authority = None

The authority of the simulated request (usually host:port)

cancel()

Cancel the pushed response and close the stream.

Returns Nothing.

get_response()

Get the pushed response provided by the server.

Returns A *HTTP20Response* object representing the pushed response.

method = None
The method of the simulated request (must be safe and cacheable, e.g. GET)

path = None
The path of the simulated request

request_headers = None
The headers the server attached to the simulated request.

scheme = None
The scheme of the simulated request

6.1.3 HTTP/1.1

class hyper.**HTTP11Connection**(*host*, *port=None*, *secure=None*, *ssl_context=None*, *proxy_host=None*, *proxy_port=None*, ***kwargs*)
An object representing a single HTTP/1.1 connection to a server.

Parameters

- **host** – The host to connect to. This may be an IP address or a hostname, and optionally may include a port: for example, `'twitter.com'`, `'twitter.com:443'` or `'127.0.0.1'`.
- **port** – (optional) The port to connect to. If not provided and one also isn't provided in the `host` parameter, defaults to 80.
- **secure** – (optional) Whether the request should use TLS. Defaults to `False` for most requests, but to `True` for any request issued to port 443.
- **ssl_context** – (optional) A class with custom certificate settings. If not provided then hyper's default `SSLContext` is used instead.
- **proxy_host** – (optional) The proxy to connect to. This can be an IP address or a host name and may include a port.
- **proxy_port** – (optional) The proxy port to connect to. If not provided and one also isn't provided in the `proxy` parameter, defaults to 8080.

close()
Closes the connection. This closes the socket and then abandons the reference to it. After calling this method, any outstanding `Response` objects will throw exceptions if attempts are made to read their bodies.

In some cases this method will automatically be called.

> **Warning:** This method should absolutely only be called when you are certain the connection object is no longer needed.

connect()
Connect to the server specified when the object was created. This is a no-op if we're already connected.

Returns Nothing.

get_response()
Returns a response object.

This is an early beta, so the response object is pretty stupid. That's ok, we'll fix it later.

network_buffer_size = None
The size of the in-memory buffer used to store data from the network. This is used as a performance optimisation. Increase buffer size to improve performance: decrease it to conserve memory. Defaults to 64kB.

parser = None
The object used to perform HTTP/1.1 parsing. Needs to conform to the standard hyper parsing interface.

request(*method*, *url*, *body=None*, *headers={}*)
This will send a request to the server using the HTTP request method `method` and the selector `url`. If the `body` argument is present, it should be string or bytes object of data to send after the headers are finished. Strings are encoded as UTF-8. To use other encodings, pass a bytes object. The Content-Length header is set to the length of the body field.

Parameters

- **method** – The request method, e.g. `'GET'`.
- **url** – The URL to contact, e.g. `'/path/segment'`.
- **body** – (optional) The request body to send. Must be a bytestring or a file-like object.
- **headers** – (optional) The headers to send on the request.

Returns Nothing.

class `hyper.`**HTTP11Response**(*code*, *reason*, *headers*, *sock*, *connection=None*)
An `HTTP11Response` wraps the HTTP/1.1 response from the server. It provides access to the response headers and the entity body. The response is an iterable object and can be used in a with statement.

close(*socket_close=False*)
Close the response. This causes the Response to lose access to the backing socket. In some cases, it can also cause the backing connection to be torn down.

Parameters **socket_close** – Whether to close the backing socket.

Returns Nothing.

headers = None
The response headers. These are determined upon creation, assigned once, and never assigned again.

read(*amt=None*, *decode_content=True*)
Reads the response body, or up to the next `amt` bytes.

Parameters

- **amt** – (optional) The amount of data to read. If not provided, all the data will be read from the response.
- **decode_content** – (optional) If `True`, will transparently decode the response data.

Returns The read data. Note that if `decode_content` is set to `True`, the actual amount of data returned may be different to the amount requested.

read_chunked(*decode_content=True*)
Reads chunked transfer encoded bodies. This method returns a generator: each iteration of which yields one chunk *unless* the chunks are compressed, in which case it yields whatever the decompressor provides for each chunk.

> **Warning:** This may yield the empty string, without that being the end of the body!

reason = None
The reason phrase returned by the server.

status = None
The status code returned by the server.

trailers = None
The response trailers. These are always intially `None`.

6.1.4 Headers

class `hyper.common.headers.`**HTTPHeaderMap**(**args*, ***kwargs*)
A structure that contains HTTP headers.

HTTP headers are a curious beast. At the surface level they look roughly like a name-value set, but in practice they have many variations that make them tricky:

- duplicate keys are allowed
- keys are compared case-insensitively
- duplicate keys are isomorphic to comma-separated values, *except when they aren't*!
- they logically contain a form of ordering

This data structure is an attempt to preserve all of that information while being as user-friendly as possible. It retains all of the mapping convenience methods (allowing by-name indexing), while avoiding using a dictionary for storage.

When iterated over, this structure returns headers in 'canonical form'. This form is a tuple, where the first entry is the header name (in lower-case), and the second entry is a list of header values (in original case).

The mapping always emits both names and values in the form of bytestrings: never unicode strings. It can accept names and values in unicode form, and will automatically be encoded to bytestrings using UTF-8. The reason for what appears to be a user-unfriendly decision here is primarily to allow the broadest-possible compatibility (to make it possible to send headers in unusual encodings) while ensuring that users are never confused about what type of data they will receive.

> **Warning:** Note that this data structure makes none of the performance guarantees of a dictionary. Lookup and deletion is not an O(1) operation. Inserting a new value *is* O(1), all other operations are O(n), including *replacing* a header entirely.

clear() → None. Remove all items from D.

get(*name*, *default=None*)
Unlike the dict get, this returns a list of items in the order they were added.

items()
This mapping iterates like the list of tuples it is.

iter_raw()
Allows iterating over the headers in 'raw' form: that is, the form in which they were added to the structure. This iteration is in order, and can be used to rebuild the original headers (e.g. to determine exactly what a server sent).

keys()
Returns an iterable of the header keys in the mapping. This explicitly does not filter duplicates, ensuring that it's the same length as len().

merge(*other*)
Merge another header set or any other dict-like into this one.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values()
This is an almost nonsensical query on a header dictionary, but we satisfy it in the exact same way we satisfy 'keys'.

6.1.5 SSLContext

`tls.`**init_context**(*cert_path=None*)
Create a new `SSLContext` that is correctly set up for an HTTP/2 connection. This SSL context object can be customized and passed as a parameter to the *`HTTPConnection`* class. Provide your own certificate file in case you don't want to use hyper's default certificate. The path to the certificate can be absolute or relative to your working directory.

Parameters **cert_path** – (optional) The path to the certificate file.

Returns An `SSLContext` correctly set up for HTTP/2.

6.1.6 Requests Transport Adapter

class `hyper.contrib.`**HTTP20Adapter**(**args*, ***kwargs*)
A Requests Transport Adapter that uses hyper to send requests over HTTP/2. This implements some degree of connection pooling to maximise the HTTP/2 gain.

add_headers(*request*, ***kwargs*)
Add any headers needed by the connection. As of v2.0 this does nothing by default, but is left for overriding by users that subclass the `HTTPAdapter`.

This should not be called from user code, and is only exposed for use when subclassing the `HTTPAdapter`.

Parameters

- **request** – The `PreparedRequest` to add headers to.
- **kwargs** – The keyword arguments from the call to send().

build_response(*request*, *resp*)
Builds a Requests' response object. This emulates most of the logic of the standard fuction but deals with the lack of the `.headers` property on the HTTP20Response object.

Additionally, this function builds in a number of features that are purely for HTTPie. This is to allow maximum compatibility with what urllib3 does, so that HTTPie doesn't fall over when it uses us.

cert_verify(*conn*, *url*, *verify*, *cert*)
Verify a SSL certificate. This method should not be called from user code, and is only exposed for use when subclassing the `HTTPAdapter`.

Parameters

- **conn** – The urllib3 connection object associated with the cert.
- **url** – The requested URL.
- **verify** – Whether we should actually verify the certificate.
- **cert** – The SSL certificate to verify.

close()
Disposes of any internal state.

Currently, this just closes the PoolManager, which closes pooled connections.

connections = None
A mapping between HTTP netlocs and `HTTP20Connection` objects.

get_connection(*host*, *port*, *scheme*)
Gets an appropriate HTTP/2 connection object based on host/port/scheme tuples.

init_poolmanager(*connections*, *maxsize*, *block=False*, ***pool_kwargs*)
Initializes a urllib3 PoolManager.

This method should not be called from user code, and is only exposed for use when subclassing the `HTTPAdapter`.

Parameters

- **connections** – The number of urllib3 connection pools to cache.
- **maxsize** – The maximum number of connections to save in the pool.
- **block** – Block when no free connections are available.
- **pool_kwargs** – Extra keyword arguments used to initialize the Pool Manager.

proxy_headers(*proxy*)
Returns a dictionary of the headers to add to any request sent through a proxy. This works with urllib3 magic to ensure that they are correctly sent to the proxy, rather than in a tunnelled request if CONNECT is being used.

This should not be called from user code, and is only exposed for use when subclassing the `HTTPAdapter`.

Parameters **proxies** – The url of the proxy being used for this request.

proxy_manager_for(*proxy*, ***proxy_kwargs*)
Return urllib3 ProxyManager for the given proxy.

This method should not be called from user code, and is only exposed for use when subclassing the `HTTPAdapter`.

Parameters

- **proxy** – The proxy to return a urllib3 ProxyManager for.
- **proxy_kwargs** – Extra keyword arguments used to configure the Proxy Manager.

Returns ProxyManager

request_url(*request*, *proxies*)
Obtain the url to use when making the final request.

If the message is being sent through a HTTP proxy, the full URL has to be used. Otherwise, we should only use the path portion of the URL.

This should not be called from user code, and is only exposed for use when subclassing the `HTTPAdapter`.

Parameters

- **request** – The `PreparedRequest` being sent.
- **proxies** – A dictionary of schemes or schemes and hosts to proxy URLs.

send(*request*, *stream=False*, ***kwargs*)
Sends a HTTP message to the server.

6.1.7 Flow Control

class `hyper.http20.window.`**BaseFlowControlManager**(*initial_window_size*, *document_size=None*)
The abstract base class for flow control managers.

This class defines the interface for pluggable flow-control managers. A flow-control manager defines a flow-control policy, which basically boils down to deciding when to increase the flow control window.

This decision can be based on a number of factors:

•the initial window size,

•the size of the document being retrieved,

•the size of the received data frames,

•any other information the manager can obtain

A flow-control manager may be defined at the connection level or at the stream level. If no stream-level flow-control manager is defined, an instance of the connection-level flow control manager is used.

A class that inherits from this one must not adjust the member variables defined in this class. They are updated and set by methods on this class.

blocked()
Called whenever the remote endpoint reports that it is blocked behind the flow control window.

When this method is called the remote endpoint is signaling that it has more data to send and that the transport layer is capable of transmitting it, but that the HTTP/2 flow control window prevents it being sent.

This method should return the size by which the window should be incremented, which may be zero. This method should *not* adjust any of the member variables of this class.

Returns The amount to increase the receive window by. Return zero if the window should not be increased.

document_size = None
The size of the document being retrieved, in bytes. This is retrieved from the Content-Length header, if provided. Note that the total number of bytes that will be received may be larger than this value due to HTTP/2 padding. It should not be assumed that simply because the the document size is smaller than the initial window size that there will never be a need to increase the window size.

increase_window_size(*frame_size*)
Determine whether or not to emit a WINDOWUPDATE frame.

This method should be overridden to determine, based on the state of the system and the size of the received frame, whether or not a WindowUpdate frame should be sent for the stream.

This method should *not* adjust any of the member variables of this class.

Note that this method is called before the window size is decremented as a result of the frame being handled.

Parameters **frame_size** – The size of the received frame. Note that this *may* be zero. When this parameter is zero, it's possible that a WINDOWUPDATE frame may want to be emitted anyway. A zero-length frame size is usually associated with a change in the size of the receive window due to a SETTINGS frame.

Returns The amount to increase the receive window by. Return zero if the window should not be increased.

initial_window_size = None
The initial size of the connection window in bytes. This is set at creation time.

window_size = None
The current size of the connection window. Any methods overridden by the user must not adjust this value.

class hyper.http20.window.**FlowControlManager**(*initial_window_size*, *document_size=None*)
hyper's default flow control manager.

This implements hyper's flow control algorithms. This algorithm attempts to reduce the number of WINDOWUPDATE frames we send without blocking the remote endpoint behind the flow control window.

This algorithm will become more complicated over time. In the current form, the algorithm is very simple:

- When the flow control window gets less than 1/4 of the maximum size, increment back to the maximum.
- Otherwise, if the flow control window gets to less than 1kB, increment back to the maximum.

6.1.8 Exceptions

class hyper.http20.exceptions.**HTTP20Error**
The base class for all of hyper's HTTP/2-related exceptions.

class hyper.http20.exceptions.**HPACKEncodingError**
An error has been encountered while performing HPACK encoding.

class hyper.http20.exceptions.**HPACKDecodingError**
An error has been encountered while performing HPACK decoding.

class hyper.http20.exceptions.**ConnectionError**
The remote party signalled an error affecting the entire HTTP/2 connection, and the connection has been closed.

h