hyper Documentation

Release 0.3.0

Cory Benfield

April 03, 2015

Contents

Release v0.3.0.

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 HTTP20Connection

conn = HTTP20Connection('http2bin.org:443')
conn.request('GET', '/get')
resp = conn.getresponse()

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 servicable 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/2 Request

With hyper installed, you can start making HTTP/2 requests. At this stage, hyper can only be used with services that *definitely* support HTTP/2. Before you begin, ensure that whichever service you're contacting definitely supports HTTP/2. 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 HTTP20Connection
>>> c = HTTP20Connection('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 HTTP20Connection.request() returns a value, unlike the equivalent http.client function. 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. 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 HTTP20Connection
>>> c = HTTP20Connection('http2bin.org')
>>> first = c.request('GET', '/get')
>>> second = c.request('POST', '/post', data='key=value')
>>> third = c.request('GET', '/ip')
>>> second_response = c.getresponse(second)
>>> first_response = c.getresponse(first)
>>> third_response = c.getresponse(third)
```

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

2.1.4 Making Your First HTTP/1.1 Request

With `hyper` installed, you can start making HTTP/2 requests. At this stage, `hyper` can only be used with services that *definitely* support HTTP/2. Before you begin, ensure that whichever service you're contacting definitely supports HTTP/2. For the rest of these examples, we'll use Twitter.

You can also use `hyper` to make HTTP/1.1 requests. The code is very similar. For example, to get the Twitter homepage:

```
>>> from hyper import HTTP11Connection
>>> c = HTTP11Connection('twitter.com:443')
>>> c.request('GET', '/')
>>> resp = c.get_response()
```

The key difference between HTTP/1.1 and HTTP/2 is that when you make HTTP/1.1 requests you do not get a stream ID. This is, of course, because HTTP/1.1 does not have streams.

Things behave exactly like they do in the HTTP/2 case, right down to the data reading:

```
>>> resp.headers['content-encoding']
[b'deflate']
>>> resp.headers
HTTPHeaderMap([(b'x-xss-protection', b'1; mode=block')...
>>> resp.status
200
>>> body = resp.read()
b'<!DOCTYPE html>\n<!--[if IE 8]><html clas ....
```

That's all it takes.

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.

All you have to do is identify a host that you'd like to communicate with over HTTP/2. Once you've worked that out, you can get started straight away:

```
>>> 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.

A quick warning: some hosts will redirect to new hostnames, which may redirect you away from HTTP/2. Make sure you install the adapter for all the hostnames you're interested in:

```
>>> a = HTTP20Adapter()
>>> s.mount('https://http2bin.org', a)
>>> s.mount('https://www.http2bin.org', a)
```

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 `HTTP20Connection` and `HTTP11Connection` as context managers:

```
with HTTP20Connection('http2bin.org') as conn:
    conn.request('GET', '/get')
    data = conn.getresponse().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 `HTTP20Connection` and `HTTP11Connection` classes are **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 `HTTP20Connection` 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` suports 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/2 Interface

6.1.2 Headers

6.1.3 Requests Transport Adapter

6.1.4 Flow Control

6.1.5 Exceptions

Python Module Index

Index