hyper-h2

Release 4.1.0

Oct 05, 2021

Contents

h2 is a HTTP/2 protocol stack, written entirely in Python. The goal of h2 is to be a common HTTP/2 stack for the Python ecosystem, usable in all programs regardless of concurrency model or environment.

To achieve this, h2 is entirely self-contained: it does no I/O of any kind, leaving that up to a wrapper library to control. This ensures that it can seamlessly work in all kinds of environments, from single-threaded code to Twisted.

Its goal is to be 100% compatible with RFC 7540, implementing a complete HTTP/2 protocol stack build on a set of finite state machines. Its secondary goals are to be fast, clear, and efficient.

For usage examples, see *Getting Started: Writing Your Own HTTP/2 Server* or consult the examples in the repository.

CHAPTER 1

Contents

1.1 Installation

h2 is a pure-python project. This means installing it is extremely simple. To get the latest release from PyPI, simply run:

```
$ python -m pip install h2
```

Alternatively, feel free to download one of the release tarballs from our GitHub page, extract it to your favourite directory, and then run

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

1.2 Getting Started: Writing Your Own HTTP/2 Server

This document explains how to get started writing fully-fledged HTTP/2 implementations using h2 as the underlying protocol stack. It covers the basic concepts you need to understand, and talks you through writing a very simple HTTP/2 server.

This document assumes you're moderately familiar with writing Python, and have *some* understanding of how computer networks work. If you don't, you'll find it a lot easier if you get some understanding of those concepts first and then return to this documentation.

1.2.1 Connections

h2's core object is the `H2Connection` object. This object is an abstract representation of the state of a single HTTP/2 connection, and holds all the important protocol state. When using h2, this object will be the first thing you create and the object that does most of the heavy lifting.

The interface to this object is relatively simple. For sending data, you call the object with methods indicating what actions you want to perform: for example, you may want to send headers (you'd use the `send_headers` method), or send data (you'd use the `send_data` method). After you've decided what actions you want to perform, you get some bytes out of the object that represent the HTTP/2-encoded representation of your actions, and send them out over the network however you see fit.

When you receive data from the network, you pass that data in to the `H2Connection` object, which returns a list of *events*. These events, covered in more detail later in *Events*, define the set of actions the remote peer has performed on the connection, as represented by the HTTP/2-encoded data you just passed to the object.

Thus, you end up with a simple loop (which you may recognise as a more-specific form of an event loop):

1. First, you perform some actions.
2. You send the data created by performing those actions to the network.
3. You read data from the network.
4. You decode those into events.
5. The events cause you to trigger some actions: go back to step 1.

Of course, HTTP/2 is more complex than that, but in the very simplest case you can write a fairly effective HTTP/2 tool using just that kind of loop. Later in this document, we'll do just that.

Some important subtleties of `H2Connection` objects are covered in *Advanced Usage*: see *Connections: Advanced* for more information. However, one subtlety should be covered, and that is this: h2's `H2Connection` object doesn't do I/O. Let's talk briefly about why.

I/O

Any useful HTTP/2 tool eventually needs to do I/O. This is because it's not very useful to be able to speak to other computers using a protocol like HTTP/2 unless you actually *speak* to them sometimes.

However, doing I/O is not a trivial thing: there are lots of different ways to do it, and once you choose a way to do it your code usually won't work well with the approaches you *didn't* choose.

While there are lots of different ways to do I/O, when it comes down to it all HTTP/2 implementations transform bytes received into events, and events into bytes to send. So there's no reason to have lots of different versions of this core protocol code: one for Twisted, one for gevent, one for threading, and one for synchronous code.

This is why we said at the top that h2 is a *HTTP/2 Protocol Stack*, not a *fully-fledged implementation*. h2 knows how to transform bytes into events and back, but that's it. The I/O and smarts might be different, but the core HTTP/2 logic is the same: that's what h2 provides.

Not doing I/O makes h2 general, and also relatively simple. It has an easy-to-understand performance envelope, it's easy to test (and as a result easy to get correct behaviour out of), and it behaves in a reproducible way. These are all great traits to have in a library that is doing something quite complex.

This document will talk you through how to build a relatively simple HTTP/2 implementation using h2, to give you an understanding of where it fits in your software.

1.2.2 Events

When writing a HTTP/2 implementation it's important to know what the remote peer is doing: if you didn't care, writing networked programs would be a lot easier!

h2 encodes the actions of the remote peer in the form of *events*. When you receive data from the remote peer and pass it into your `H2Connection` object (see *Connections*), the `H2Connection` returns a list of objects, each one representing a single event that has occurred. Each event refers to a single action the remote peer has taken.

Some events are fairly high-level, referring to things that are more general than HTTP/2: for example, the `RequestReceived` event is a general HTTP concept, not just a HTTP/2 one. Other events are extremely HTTP/2-specific: for example, `PushedStreamReceived` refers to Server Push, a very HTTP/2-specific concept.

The reason these events exist is that h2 is intended to be very general. This means that, in many cases, h2 does not know exactly what to do in response to an event. Your code will need to handle these events, and make decisions about what to do. That's the major role of any HTTP/2 implementation built on top of h2.

A full list of events is available in *Events*. For the purposes of this example, we will handle only a small set of events.

1.2.3 Writing Your Server

Armed with the knowledge you just obtained, we're going to write a very simple HTTP/2 web server. The goal of this server is to write a server that can handle a HTTP GET, and that returns the headers sent by the client, encoded in JSON. Basically, something a lot like httpbin.org/get. Nothing fancy, but this is a good way to get a handle on how you should interact with h2.

For the sake of simplicity, we're going to write this using the Python standard library, in Python 3. In reality, you'll probably want to use an asynchronous framework of some kind: see the examples directory in the repository for some examples of how you'd do that.

Before we start, create a new file called `h2server.py`: we'll use that as our workspace. Additionally, you should install h2: follow the instructions in *Installation*.

Step 1: Sockets

To begin with, we need to make sure we can listen for incoming data and send it back. To do that, we need to use the standard library's socket module. For now we're going to skip doing TLS: if you want to reach your server from your web browser, though, you'll need to add TLS and some other function. Consider looking at our examples in our examples directory instead.

Let's begin. First, open up `h2server.py`. We need to import the socket module and start listening for connections.

This is not a socket tutorial, so we're not going to dive too deeply into how this works. If you want more detail about sockets, there are lots of good tutorials on the web that you should investigate.

When you want to listen for incoming connections, the you need to *bind* an address first. So let's do that. Try setting up your file to look like this:

```
import socket

sock = socket.socket()
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind(('0.0.0.0', 8080))
sock.listen(5)

while True:
    print(sock.accept())
```

In a shell window, execute this program (`python h2server.py`). Then, open another shell and run `curl http://localhost:8080/`. In the first shell, you should see something like this:

```
$ python h2server.py
(<socket.socket fd=4, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM,␣
↪proto=0, laddr=('127.0.0.1', 8080), raddr=('127.0.0.1', 58800)>, ('127.0.0.1',␣
↪58800))
```

Run that curl command a few more times. You should see a few more similar lines appear. Note that the curl command itself will exit with an error. That's fine: it happens because we didn't send any data.

Now go ahead and stop the server running by hitting Ctrl+C in the first shell. You should see a KeyboardInterrupt error take the process down.

What's the program above doing? Well, first it creates a socket object. This socket is then *bound* to a specific address: ('0.0.0.0', 8080). This is a special address: it means that this socket should be listening for any traffic to TCP port 8080. Don't worry about the call to setsockopt: it just makes sure you can run this program repeatedly.

We then loop forever calling the accept method on the socket. The accept method blocks until someone attempts to connect to our TCP port: when they do, it returns a tuple: the first element is a new socket object, the second element is a tuple of the address the new connection is from. You can see this in the output from our h2server.py script.

At this point, we have a script that can accept inbound connections. This is a good start! Let's start getting HTTP/2 involved.

Step 2: Add a H2Connection

Now that we can listen for socket information, we want to prepare our HTTP/2 connection object and start handing it data. For now, let's just see what happens as we feed it data.

To make HTTP/2 connections, we need a tool that knows how to speak HTTP/2. Most versions of curl in the wild don't, so let's install a Python tool. In your Python environment, run pip install hyper. This will install a Python command-line HTTP/2 tool called hyper. To confirm that it works, try running this command and verifying that the output looks similar to the one shown below:

```
$ hyper GET https://nghttp2.org/httpbin/get
{'args': {},
 'headers': {'Host': 'nghttp2.org'},
 'origin': '10.0.0.2',
 'url': 'https://nghttp2.org/httpbin/get'}
```

Assuming it works, you're now ready to start sending HTTP/2 data.

Back in our h2server.py script, we're going to want to start handling data. Let's add a function that takes a socket returned from accept, and reads data from it. Let's call that function handle. That function should create a *H2Connection* object and then loop on the socket, reading data and passing it to the connection.

To read data from a socket we need to call recv. The recv function takes a number as its argument, which is the *maximum* amount of data to be returned from a single call (note that recv will return as soon as any data is available, even if that amount is vastly less than the number you passed to it). For the purposes of writing this kind of software the specific value is not enormously useful, but should not be overly large. For that reason, when you're unsure, a number like 4096 or 65535 is a good bet. We'll use 65535 for this example.

The function should look something like this:

```python
import h2.connection
import h2.config

def handle(sock):
    config = h2.config.H2Configuration(client_side=False)
    conn = h2.connection.H2Connection(config=config)

    while True:
        data = sock.recv(65535)
        print(conn.receive_data(data))
```

Let's update our main loop so that it passes data on to our new data handling function. Your h2server.py should end up looking a like this:

```
import socket

import h2.connection
import h2.config

def handle(sock):
    config = h2.config.H2Configuration(client_side=False)
    conn = h2.connection.H2Connection(config=config)

    while True:
        data = sock.recv(65535)
        if not data:
            break

        print(conn.receive_data(data))


sock = socket.socket()
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind(('0.0.0.0', 8080))
sock.listen(5)

while True:
    handle(sock.accept()[0])
```

Running that in one shell, in your other shell you can run hyper --h2 GET http://localhost:8080/. That shell should hang, and you should then see the following output from your h2server.py shell:

```
$ python h2server.py
[<h2.events.RemoteSettingsChanged object at 0x10c4ee390>]
```

You'll then need to kill hyper and h2server.py with Ctrl+C. Feel free to do this a few times, to see how things behave.

So, what did we see here? When the connection was opened, we used the recv method to read some data from the socket, in a loop. We then passed that data to the connection object, which returned us a single event object: *RemoteSettingsChanged*.

But what we didn't see was anything else. So it seems like all hyper did was change its settings, but nothing else. If you look at the other hyper window, you'll notice that it hangs for a while and then eventually fails with a socket timeout. It was waiting for something: what?

Well, it turns out that at the start of a connection, both sides need to send a bit of data, called "the HTTP/2 preamble". We don't need to get into too much detail here, but basically both sides need to send a single block of HTTP/2 data that tells the other side what their settings are. hyper did that, but we didn't.

Let's do that next.

Step 3: Sending the Preamble

h2 makes doing connection setup really easy. All you need to do is call the *initiate_connection* method, and then send the corresponding data. Let's update our handle function to do just that:

```
def handle(sock):
    config = h2.config.H2Configuration(client_side=False)
    conn = h2.connection.H2Connection(config=config)
    conn.initiate_connection()
    sock.sendall(conn.data_to_send())

    while True:
        data = sock.recv(65535)
        print(conn.receive_data(data))
```

The big change here is the call to `initiate_connection`, but there's another new method in there: *`data_to_send`*.

When you make function calls on your `H2Connection` object, these will often want to cause HTTP/2 data to be written out to the network. But h2 doesn't do any I/O, so it can't do that itself. Instead, it writes it to an internal buffer. You can retrieve data from this buffer using the `data_to_send` method. There are some subtleties about that method, but we don't need to worry about them right now: all we need to do is make sure we're sending whatever data is outstanding.

Your `h2server.py` script should now look like this:

```
import socket

import h2.connection
import h2.config

def handle(sock):
    config = h2.config.H2Configuration(client_side=False)
    conn = h2.connection.H2Connection(config=config)
    conn.initiate_connection()
    sock.sendall(conn.data_to_send())

    while True:
        data = sock.recv(65535)
        if not data:
            break

        print(conn.receive_data(data))


sock = socket.socket()
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind(('0.0.0.0', 8080))
sock.listen(5)

while True:
    handle(sock.accept()[0])
```

With this change made, rerun your `h2server.py` script and hit it with the same `hyper` command: `hyper --h2 GET http://localhost:8080/`. The `hyper` command still hangs, but this time we get a bit more output from our `h2server.py` script:

```
$ python h2server.py
[<h2.events.RemoteSettingsChanged object at 0x10292d390>]
[<h2.events.SettingsAcknowledged object at 0x102b3a160>]
[<h2.events.RequestReceived object at 0x102b3a3c8>, <h2.events.StreamEnded object at␣
↪0x102b3a400>]
```

So, what's happening?

The first thing to note is that we're going around our loop more than once now. First, we receive some data that triggers a *RemoteSettingsChanged* event. Then, we get some more data that triggers a *SettingsAcknowledged* event. Finally, even more data that triggers *two* events: *RequestReceived* and *StreamEnded*.

So, what's happening is that `hyper` is telling us about its settings, acknowledging ours, and then sending us a request. Then it ends a *stream*, which is a HTTP/2 communications channel that holds a request and response pair.

A stream isn't done until it's either *reset* or both sides *close* it: in this sense it's bi-directional. So what the `StreamEnded` event tells us is that `hyper` is closing its half of the stream: it won't send us any more data on that stream. That means the request is done.

So why is `hyper` hanging? Well, we haven't sent a response yet: let's do that.

Step 4: Handling Events

What we want to do is send a response when we receive a request. Happily, we get an event when we receive a request, so we can use that to be our signal.

Let's define a new function that sends a response. For now, this response can just be a little bit of data that prints "it works!".

The function should take the `H2Connection` object, and the event that signaled the request. Let's define it.

```
def send_response(conn, event):
    stream_id = event.stream_id
    conn.send_headers(
        stream_id=stream_id,
        headers=[
            (':status', '200'),
            ('server', 'basic-h2-server/1.0')
        ],
    )
    conn.send_data(
        stream_id=stream_id,
        data=b'it works!',
        end_stream=True
    )
```

So while this is only a short function, there's quite a lot going on here we need to unpack. Firstly, what's a stream ID? Earlier we discussed streams briefly, to say that they're a bi-directional communications channel that holds a request and response pair. Part of what makes HTTP/2 great is that there can be lots of streams going on at once, sending and receiving different requests and responses. To identify each stream, we use a *stream ID*. These are unique across the lifetime of a connection, and they go in ascending order.

Most `H2Connection` functions take a stream ID: they require you to actively tell the connection which one to use. In this case, as a simple server, we will never need to choose a stream ID ourselves: the client will always choose one for us. That means we'll always be able to get the one we need off the events that fire.

Next, we send some *headers*. In HTTP/2, a response is made up of some set of headers, and optionally some data. The headers have to come first: if you're a client then you'll be sending *request* headers, but in our case these headers are our *response* headers.

Mostly these aren't very exciting, but you'll notice once special header in there: `:status`. This is a HTTP/2-specific header, and it's used to hold the HTTP status code that used to go at the top of a HTTP response. Here, we're saying the response is `200 OK`, which is successful.

To send headers in h2, you use the *send_headers* function.

Next, we want to send the body data. To do that, we use the `send_data` function. This also takes a stream ID. Note that the data is binary: h2 does not work with unicode strings, so you *must* pass bytestrings to the `H2Connection`. The one exception is headers: h2 will automatically encode those into UTF-8.

The last thing to note is that on our call to `send_data`, we set `end_stream` to `True`. This tells h2 (and the remote peer) that we're done with sending data: the response is over. Because we know that `hyper` will have ended its side of the stream, when we end ours the stream will be totally done with.

We're nearly ready to go with this: we just need to plumb this function in. Let's amend our `handle` function again:

```
import h2.events
import h2.config

def handle(sock):
    config = h2.config.H2Configuration(client_side=False)
    conn = h2.connection.H2Connection(config=config)
    conn.initiate_connection()
    sock.sendall(conn.data_to_send())

    while True:
        data = sock.recv(65535)
        if not data:
            break

        events = conn.receive_data(data)
        for event in events:
            if isinstance(event, h2.events.RequestReceived):
                send_response(conn, event)

        data_to_send = conn.data_to_send()
        if data_to_send:
            sock.sendall(data_to_send)
```

The changes here are all at the end. Now, when we receive some events, we look through them for the `RequestReceived` event. If we find it, we make sure we send a response.

Then, at the bottom of the loop we check whether we have any data to send, and if we do, we send it. Then, we repeat again.

With these changes, your `h2server.py` file should look like this:

```
import socket

import h2.connection
import h2.events
import h2.config

def send_response(conn, event):
    stream_id = event.stream_id
    conn.send_headers(
        stream_id=stream_id,
        headers=[
            (':status', '200'),
            ('server', 'basic-h2-server/1.0')
        ],
    )
    conn.send_data(
        stream_id=stream_id,
        data=b'it works!',
```

(continues on next page)

(continued from previous page)

```
        end_stream=True
    )

def handle(sock):
    config = h2.config.H2Configuration(client_side=False)
    conn = h2.connection.H2Connection(config=config)
    conn.initiate_connection()
    sock.sendall(conn.data_to_send())

    while True:
        data = sock.recv(65535)
        if not data:
            break

        events = conn.receive_data(data)
        for event in events:
            if isinstance(event, h2.events.RequestReceived):
                send_response(conn, event)

        data_to_send = conn.data_to_send()
        if data_to_send:
            sock.sendall(data_to_send)


sock = socket.socket()
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind(('0.0.0.0', 8080))
sock.listen(5)

while True:
    handle(sock.accept()[0])
```

Alright. Let's run this, and then run our `hyper` command again.

This time, nothing is printed from our server, and the `hyper` side prints `it works!`. Success! Try running it a few more times, and we can see that not only does it work the first time, it works the other times too!

We can speak HTTP/2! Let's add the final step: returning the JSON-encoded request headers.

Step 5: Returning Headers

If we want to return the request headers in JSON, the first thing we have to do is find them. Handily, if you check the documentation for *RequestReceived* you'll find that this event carries, in addition to the stream ID, the request headers.

This means we can make a really simple change to our `send_response` function to take those headers and encode them as a JSON object. Let's do that:

```
import json

def send_response(conn, event):
    stream_id = event.stream_id
    response_data = json.dumps(dict(event.headers)).encode('utf-8')

    conn.send_headers(
        stream_id=stream_id,
```

(continues on next page)

(continued from previous page)

```
        headers=[
            (':status', '200'),
            ('server', 'basic-h2-server/1.0'),
            ('content-length', str(len(response_data))),
            ('content-type', 'application/json'),
        ],
    )
    conn.send_data(
        stream_id=stream_id,
        data=response_data,
        end_stream=True
    )
```

This is a really simple change, but it's all we need to do: a few extra headers and the JSON dump, but that's it.

Section 6: Bringing It All Together

This should be all we need!

Let's take all the work we just did and throw that into our `h2server.py` file, which should now look like this:

```
import json
import socket

import h2.connection
import h2.events
import h2.config

def send_response(conn, event):
    stream_id = event.stream_id
    response_data = json.dumps(dict(event.headers)).encode('utf-8')

    conn.send_headers(
        stream_id=stream_id,
        headers=[
            (':status', '200'),
            ('server', 'basic-h2-server/1.0'),
            ('content-length', str(len(response_data))),
            ('content-type', 'application/json'),
        ],
    )
    conn.send_data(
        stream_id=stream_id,
        data=response_data,
        end_stream=True
    )

def handle(sock):
    config = h2.config.H2Configuration(client_side=False)
    conn = h2.connection.H2Connection(config=config)
    conn.initiate_connection()
    sock.sendall(conn.data_to_send())

    while True:
        data = sock.recv(65535)
        if not data:
```

(continues on next page)

(continued from previous page)

```
            break

        events = conn.receive_data(data)
        for event in events:
            if isinstance(event, h2.events.RequestReceived):
                send_response(conn, event)

        data_to_send = conn.data_to_send()
        if data_to_send:
            sock.sendall(data_to_send)


sock = socket.socket()
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind(('0.0.0.0', 8080))
sock.listen(5)

while True:
    handle(sock.accept()[0])
```

Now, execute `h2server.py` and then point `hyper` at it again. You should see something like the following output from `hyper`:

```
$ hyper --h2 GET http://localhost:8080/
{":scheme": "http", ":authority": "localhost", ":method": "GET", ":path": "/"}
```

Here you can see the HTTP/2 request 'special headers' that `hyper` sends. These are similar to the `:status` header we have to send on our response: they encode important parts of the HTTP request in a clearly-defined way. If you were writing a client stack using h2, you'd need to make sure you were sending those headers.

Congratulations!

Congratulations! You've written your first HTTP/2 server! If you want to extend it, there are a few directions you could investigate:

- We didn't handle a few events that we saw were being raised: you could add some methods to handle those appropriately.
- Right now our server is single threaded, so it can only handle one client at a time. Consider rewriting this server to use threads, or writing this server again using your favourite asynchronous programming framework.

  If you plan to use threads, you should know that a `H2Connection` object is deliberately not thread-safe. As a possible design pattern, consider creating threads and passing the sockets returned by `accept` to those threads, and then letting those threads create their own `H2Connection` objects.
- Take a look at some of our long-form code examples in *Code Examples*.
- Alternatively, try playing around with our examples in our repository's examples directory. These examples are a bit more fully-featured, and can be reached from your web browser. Try adjusting what they do, or adding new features to them!
- You may want to make this server reachable from your web browser. To do that, you'll need to add proper TLS support to your server. This can be tricky, and in many cases requires PyOpenSSL in addition to the other libraries you have installed. Check the Eventlet example to see what PyOpenSSL code is required to TLS-ify your server.

1.3 Negotiating HTTP/2

RFC 7540 specifies three methods of negotiating HTTP/2 connections. This document outlines how to use h2 with each one.

1.3.1 HTTPS URLs (ALPN)

Starting HTTP/2 for HTTPS URLs is outlined in RFC 7540 Section 3.3. In this case, the client and server use a TLS extension to negotiate HTTP/2: ALPN. How to use ALPN is currently not covered in this document: please consult the documentation for either the `ssl module` in the standard library, or the `PyOpenSSL` third-party modules, for more on this topic.

This method is the simplest to use once the TLS connection is established. To use it with h2, after you've established the connection and confirmed that HTTP/2 has been negotiated with ALPN, create a *`H2Connection`* object and call *`H2Connection.initiate_connection`*. This will ensure that the appropriate preamble data is placed in the data buffer. You should then immediately send the data returned by *`H2Connection.data_to_send`* on your TLS connection.

At this point, you're free to use all the HTTP/2 functionality provided by h2.

Note: Although h2 is not concerned with negotiating protocol versions, it is important to note that support for ALPN is not available in the standard library of Python versions < 2.7.9. As a consequence, clients may encounter various errors due to protocol versions mismatch.

Server Setup Example

This example uses the APIs as defined in Python 3.5. If you are using an older version of Python you may not have access to the APIs used here. As noted above, please consult the documentation for the `ssl module` to confirm.

```python
# -*- coding: utf-8 -*-
"""
Server HTTPS Setup
~~~~~~~~~~~~~~~~~~

This example code fragment demonstrates how to set up a HTTP/2 server that
negotiates HTTP/2 using NPN and ALPN. For the sake of maximum explanatory value
this code uses the synchronous, low-level sockets API: however, if you're not
using sockets directly (e.g. because you're using asyncio), you should focus on
the set up required for the SSLContext object. For other concurrency libraries
you may need to use other setup (e.g. for Twisted you'll need to use
IProtocolNegotiationFactory).

This code requires Python 3.5 or later.
"""
import h2.config
import h2.connection
import socket
import ssl


def establish_tcp_connection():
    """
    This function establishes a server-side TCP connection. How it works isn't
```

(continues on next page)

(continued from previous page)

```
    very important to this example.
    """
    bind_socket = socket.socket()
    bind_socket.bind(('', 443))
    bind_socket.listen(5)
    return bind_socket.accept()[0]


def get_http2_ssl_context():
    """
    This function creates an SSLContext object that is suitably configured for
    HTTP/2. If you're working with Python TLS directly, you'll want to do the
    exact same setup as this function does.
    """
    # Get the basic context from the standard library.
    ctx = ssl.create_default_context(purpose=ssl.Purpose.CLIENT_AUTH)

    # RFC 7540 Section 9.2: Implementations of HTTP/2 MUST use TLS version 1.2
    # or higher. Disable TLS 1.1 and lower.
    ctx.options |= (
        ssl.OP_NO_SSLv2 | ssl.OP_NO_SSLv3 | ssl.OP_NO_TLSv1 | ssl.OP_NO_TLSv1_1
    )

    # RFC 7540 Section 9.2.1: A deployment of HTTP/2 over TLS 1.2 MUST disable
    # compression.
    ctx.options |= ssl.OP_NO_COMPRESSION

    # RFC 7540 Section 9.2.2: "deployments of HTTP/2 that use TLS 1.2 MUST
    # support TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256". In practice, the
    # blocklist defined in this section allows only the AES GCM and ChaCha20
    # cipher suites with ephemeral key negotiation.
    ctx.set_ciphers("ECDHE+AESGCM:ECDHE+CHACHA20:DHE+AESGCM:DHE+CHACHA20")

    # We want to negotiate using NPN and ALPN. ALPN is mandatory, but NPN may
    # be absent, so allow that. This setup allows for negotiation of HTTP/1.1.
    ctx.set_alpn_protocols(["h2", "http/1.1"])

    try:
        ctx.set_npn_protocols(["h2", "http/1.1"])
    except NotImplementedError:
        pass

    return ctx


def negotiate_tls(tcp_conn, context):
    """
    Given an established TCP connection and a HTTP/2-appropriate TLS context,
    this function:

    1. wraps TLS around the TCP connection.
    2. confirms that HTTP/2 was negotiated and, if it was not, throws an error.
    """
    tls_conn = context.wrap_socket(tcp_conn, server_side=True)

    # Always prefer the result from ALPN to that from NPN.
    # You can only check what protocol was negotiated once the handshake is
```

(continues on next page)

(continued from previous page)

```
    # complete.
    negotiated_protocol = tls_conn.selected_alpn_protocol()
    if negotiated_protocol is None:
        negotiated_protocol = tls_conn.selected_npn_protocol()

    if negotiated_protocol != "h2":
        raise RuntimeError("Didn't negotiate HTTP/2!")

    return tls_conn


def main():
    # Step 1: Set up your TLS context.
    context = get_http2_ssl_context()

    # Step 2: Receive a TCP connection.
    connection = establish_tcp_connection()

    # Step 3: Wrap the connection in TLS and validate that we negotiated HTTP/2
    tls_connection = negotiate_tls(connection, context)

    # Step 4: Create a server-side H2 connection.
    config = h2.config.H2Configuration(client_side=False)
    http2_connection = h2.connection.H2Connection(config=config)

    # Step 5: Initiate the connection
    http2_connection.initiate_connection()
    tls_connection.sendall(http2_connection.data_to_send())

    # The TCP, TLS, and HTTP/2 handshakes are now complete. You can enter your
    # main loop now.
```

Client Setup Example

The client example is very similar to the server example above. The `SSLContext` object requires some minor changes, as does the *H2Connection*, but the bulk of the code is the same.

```
# -*- coding: utf-8 -*-
"""
Client HTTPS Setup
~~~~~~~~~~~~~~~~~~

This example code fragment demonstrates how to set up a HTTP/2 client that
negotiates HTTP/2 using NPN and ALPN. For the sake of maximum explanatory value
this code uses the synchronous, low-level sockets API: however, if you're not
using sockets directly (e.g. because you're using asyncio), you should focus on
the set up required for the SSLContext object. For other concurrency libraries
you may need to use other setup (e.g. for Twisted you'll need to use
IProtocolNegotiationFactory).

This code requires Python 3.5 or later.
"""
import h2.connection
import socket
import ssl
```

(continues on next page)

(continued from previous page)

```


def establish_tcp_connection():
    """
    This function establishes a client-side TCP connection. How it works isn't
    very important to this example. For the purpose of this example we connect
    to localhost.
    """
    return socket.create_connection(('localhost', 443))


def get_http2_ssl_context():
    """
    This function creates an SSLContext object that is suitably configured for
    HTTP/2. If you're working with Python TLS directly, you'll want to do the
    exact same setup as this function does.
    """
    # Get the basic context from the standard library.
    ctx = ssl.create_default_context(purpose=ssl.Purpose.SERVER_AUTH)

    # RFC 7540 Section 9.2: Implementations of HTTP/2 MUST use TLS version 1.2
    # or higher. Disable TLS 1.1 and lower.
    ctx.options |= (
        ssl.OP_NO_SSLv2 | ssl.OP_NO_SSLv3 | ssl.OP_NO_TLSv1 | ssl.OP_NO_TLSv1_1
    )

    # RFC 7540 Section 9.2.1: A deployment of HTTP/2 over TLS 1.2 MUST disable
    # compression.
    ctx.options |= ssl.OP_NO_COMPRESSION

    # RFC 7540 Section 9.2.2: "deployments of HTTP/2 that use TLS 1.2 MUST
    # support TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256". In practice, the
    # blocklist defined in this section allows only the AES GCM and ChaCha20
    # cipher suites with ephemeral key negotiation.
    ctx.set_ciphers("ECDHE+AESGCM:ECDHE+CHACHA20:DHE+AESGCM:DHE+CHACHA20")

    # We want to negotiate using NPN and ALPN. ALPN is mandatory, but NPN may
    # be absent, so allow that. This setup allows for negotiation of HTTP/1.1.
    ctx.set_alpn_protocols(["h2", "http/1.1"])

    try:
        ctx.set_npn_protocols(["h2", "http/1.1"])
    except NotImplementedError:
        pass

    return ctx


def negotiate_tls(tcp_conn, context):
    """
    Given an established TCP connection and a HTTP/2-appropriate TLS context,
    this function:

    1. wraps TLS around the TCP connection.
    2. confirms that HTTP/2 was negotiated and, if it was not, throws an error.
    """
    # Note that SNI is mandatory for HTTP/2, so you *must* pass the
```

(continues on next page)

(continued from previous page)

```python
    # server_hostname argument.
    tls_conn = context.wrap_socket(tcp_conn, server_hostname='localhost')

    # Always prefer the result from ALPN to that from NPN.
    # You can only check what protocol was negotiated once the handshake is
    # complete.
    negotiated_protocol = tls_conn.selected_alpn_protocol()
    if negotiated_protocol is None:
        negotiated_protocol = tls_conn.selected_npn_protocol()

    if negotiated_protocol != "h2":
        raise RuntimeError("Didn't negotiate HTTP/2!")

    return tls_conn


def main():
    # Step 1: Set up your TLS context.
    context = get_http2_ssl_context()

    # Step 2: Create a TCP connection.
    connection = establish_tcp_connection()

    # Step 3: Wrap the connection in TLS and validate that we negotiated HTTP/2
    tls_connection = negotiate_tls(connection, context)

    # Step 4: Create a client-side H2 connection.
    http2_connection = h2.connection.H2Connection()

    # Step 5: Initiate the connection
    http2_connection.initiate_connection()
    tls_connection.sendall(http2_connection.data_to_send())

    # The TCP, TLS, and HTTP/2 handshakes are now complete. You can enter your
    # main loop now.
```

1.3.2 HTTP URLs (Upgrade)

Starting HTTP/2 for HTTP URLs is outlined in RFC 7540 Section 3.2. In this case, the client and server use the HTTP Upgrade mechanism originally described in RFC 7230 Section 6.7. The client sends its initial HTTP/1.1 request with two extra headers. The first is `Upgrade: h2c`, which requests upgrade to cleartext HTTP/2. The second is a `HTTP2-Settings` header, which contains a specially formatted string that encodes a HTTP/2 Settings frame.

To do this with h2 you have two slightly different flows: one for clients, one for servers.

Clients

For a client, when sending the first request you should manually add your `Upgrade` header. You should then create a *`H2Connection`* object and call *`H2Connection.initiate_upgrade_connection`* with no arguments. This method will return a bytestring to use as the value of your `HTTP2-Settings` header.

If the server returns a `101` status code, it has accepted the upgrade, and you should immediately send the data returned by *`H2Connection.data_to_send`*. Now you should consume the entire `101` header block. All data after the `101` header block is HTTP/2 data that should be fed directly to *`H2Connection.receive_data`* and handled as normal with h2.

If the server does not return a `101` status code then it is not upgrading. Continue with HTTP/1.1 as normal: you may throw away your `H2Connection` object, as it is of no further use.

The server will respond to your original request in HTTP/2. Please pay attention to the events received from h2, as they will define the server's response.

Client Example

The code below demonstrates how to handle a plaintext upgrade from the perspective of the client. For the purposes of keeping the example code as simple and generic as possible it uses the synchronous socket API that comes with the Python standard library: if you want to use asynchronous I/O, you will need to translate this code to the appropriate idiom.

```python
# -*- coding: utf-8 -*-
"""
Client Plaintext Upgrade
~~~~~~~~~~~~~~~~~~~~~~~~

This example code fragment demonstrates how to set up a HTTP/2 client that uses
the plaintext HTTP Upgrade mechanism to negotiate HTTP/2 connectivity. For
maximum explanatory value it uses the synchronous socket API that comes with
the Python standard library. In product code you will want to use an actual
HTTP/1.1 client if possible.

This code requires Python 3.5 or later.
"""
import h2.connection
import socket


def establish_tcp_connection():
    """
    This function establishes a client-side TCP connection. How it works isn't
    very important to this example. For the purpose of this example we connect
    to localhost.
    """
    return socket.create_connection(('localhost', 80))


def send_initial_request(connection, settings):
    """
    For the sake of this upgrade demonstration, we're going to issue a GET
    request against the root of the site. In principle the best request to
    issue for an upgrade is actually ``OPTIONS *``, but this is remarkably
    poorly supported and can break in weird ways.
    """
    # Craft our initial request per RFC 7540 Section 3.2. This requires two
    # special header fields: the Upgrade headre, and the HTTP2-Settings header.
    # The value of the HTTP2-Settings header field comes from h2.
    request = (
        b"GET / HTTP/1.1\r\n" +
        b"Host: localhost\r\n" +
        b"Upgrade: h2c\r\n" +
        b"HTTP2-Settings: " + settings + b"\r\n" +
        b"\r\n"
    )
    connection.sendall(request)
```

(continues on next page)

(continued from previous page)

```python


def get_upgrade_response(connection):
    """
    This function reads from the socket until the HTTP/1.1 end-of-headers
    sequence (CRLFCRLF) is received. It then checks what the status code of the
    response is.

    This is not a substitute for proper HTTP/1.1 parsing, but it's good enough
    for example purposes.
    """
    data = b''
    while b'\r\n\r\n' not in data:
        data += connection.recv(8192)

    headers, rest = data.split(b'\r\n\r\n', 1)

    # An upgrade response begins HTTP/1.1 101 Switching Protocols. Look for the
    # code. In production code you should also check that the upgrade is to
    # h2c, but here we know we only offered one upgrade so there's only one
    # possible upgrade in use.
    split_headers = headers.split()
    if split_headers[1] != b'101':
        raise RuntimeError("Not upgrading!")

    # We don't care about the HTTP/1.1 data anymore, but we do care about
    # any other data we read from the socket: this is going to be HTTP/2 data
    # that must be passed to the H2Connection.
    return rest


def main():
    """
    The client upgrade flow.
    """
    # Step 1: Establish the TCP connecton.
    connection = establish_tcp_connection()

    # Step 2: Create H2 Connection object, put it in upgrade mode, and get the
    # value of the HTTP2-Settings header we want to use.
    h2_connection = h2.connection.H2Connection()
    settings_header_value = h2_connection.initiate_upgrade_connection()

    # Step 3: Send the initial HTTP/1.1 request with the upgrade fields.
    send_initial_request(connection, settings_header_value)

    # Step 4: Read the HTTP/1.1 response, look for 101 response.
    extra_data = get_upgrade_response(connection)

    # Step 5: Immediately send the pending HTTP/2 data.
    connection.sendall(h2_connection.data_to_send())

    # Step 6: Feed the body data to the connection.
    events = connection.receive_data(extra_data)

    # Now you can enter your main loop, beginning by processing the first set
    # of events above. These events may include ResponseReceived, which will
```

(continues on next page)

(continued from previous page)

```
    # contain the response to the request we made in Step 3.
    main_loop(events)
```

Servers

If the first request you receive on a connection from the client contains an `Upgrade` header with the `h2c` token in it, and you're willing to upgrade, you should create a *H2Connection* object and call *H2Connection.initiate_upgrade_connection* with the value of the `HTTP2-Settings` header (as a bytestring) as the only argument.

Then, you should send back a `101` response that contains `h2c` in the `Upgrade` header. That response will inform the client that you're switching to HTTP/2. Then, you should immediately send the data that is returned to you by *H2Connection.data_to_send* on the connection: this is a necessary part of the HTTP/2 upgrade process.

At this point, you may now respond to the original HTTP/1.1 request in HTTP/2 by calling the appropriate methods on the *H2Connection* object. No further HTTP/1.1 may be sent on this connection: from this point onward, all data sent by you and the client will be HTTP/2 data.

Server Example

The code below demonstrates how to handle a plaintext upgrade from the perspective of the server. For the purposes of keeping the example code as simple and generic as possible it uses the synchronous socket API that comes with the Python standard library: if you want to use asynchronous I/O, you will need to translate this code to the appropriate idiom.

```
# -*- coding: utf-8 -*-
"""
Server Plaintext Upgrade
~~~~~~~~~~~~~~~~~~~~~~~~

This example code fragment demonstrates how to set up a HTTP/2 server that uses
the plaintext HTTP Upgrade mechanism to negotiate HTTP/2 connectivity. For
maximum explanatory value it uses the synchronous socket API that comes with
the Python standard library. In product code you will want to use an actual
HTTP/1.1 server library if possible.

This code requires Python 3.5 or later.
"""
import h2.config
import h2.connection
import re
import socket


def establish_tcp_connection():
    """
    This function establishes a server-side TCP connection. How it works isn't
    very important to this example.
    """
    bind_socket = socket.socket()
    bind_socket.bind(('', 443))
    bind_socket.listen(5)
    return bind_socket.accept()[0]
```

(continues on next page)

(continued from previous page)

```python


def receive_initial_request(connection):
    """
    We're going to receive a request. For the sake of this example, we're going
    to assume that the first request has no body. If it doesn't have the
    Upgrade: h2c header field and the HTTP2-Settings header field, we'll throw
    errors.

    In production code, you should use a proper HTTP/1.1 parser and actually
    serve HTTP/1.1 requests!

    Returns the value of the HTTP2-Settings header field.
    """
    data = b''
    while not data.endswith(b'\r\n\r\n'):
        data += connection.recv(8192)

    match = re.search(b'Upgrade: h2c\r\n', data)
    if match is None:
        raise RuntimeError("HTTP/2 upgrade not requested!")

    # We need to look for the HTTP2-Settings header field. Again, in production
    # code you shouldn't use regular expressions for this, but it's good enough
    # for the example.
    match = re.search(b'HTTP2-Settings: (\\S+)\r\n', data)
    if match is None:
        raise RuntimeError("HTTP2-Settings header field not present!")

    return match.group(1)


def send_upgrade_response(connection):
    """
    This function writes the 101 Switching Protocols response.
    """
    response = (
        b"HTTP/1.1 101 Switching Protocols\r\n"
        b"Upgrade: h2c\r\n"
        b"\r\n"
    )
    connection.sendall(response)


def main():
    """
    The server upgrade flow.
    """
    # Step 1: Establish the TCP connecton.
    connection = establish_tcp_connection()

    # Step 2: Read the response. We expect this to request an upgrade.
    settings_header_value = receive_initial_request(connection)

    # Step 3: Create a H2Connection object in server mode, and pass it the
    # value of the HTTP2-Settings header field.
    config = h2.config.H2Configuration(client_side=False)
```

(continues on next page)

(continued from previous page)

```
    h2_connection = h2.connection.H2Connection(config=config)
    h2_connection.initiate_upgrade_connection(
        settings_header=settings_header_value
    )

    # Step 4: Send the 101 Switching Protocols response.
    send_upgrade_response(connection)

    # Step 5: Send pending HTTP/2 data.
    connection.sendall(h2_connection.data_to_send())

    # At this point, you can enter your main loop. The first step has to be to
    # send the response to the initial HTTP/1.1 request you received on stream
    # 1.
    main_loop()
```

1.3.3 Prior Knowledge

It’s possible that you as a client know that a particular server supports HTTP/2, and that you do not need to perform any of the negotiations described above. In that case, you may follow the steps in *HTTPS URLs (ALPN)*, ignoring all references to ALPN: there’s no need to perform the upgrade dance described in *HTTP URLs (Upgrade)*.

1.4 Code Examples

This section of the documentation contains long-form code examples. These are intended as references for developers that would like to get an understanding of how h2 fits in with various Python I/O frameworks.

1.4.1 Example Servers

Asyncio Example Server

This example is a basic HTTP/2 server written using asyncio, using some functionality that was introduced in Python 3.5. This server represents basically just the same JSON-headers-returning server that was built in the *Getting Started: Writing Your Own HTTP/2 Server* document.

This example demonstrates some basic asyncio techniques.

```
# -*- coding: utf-8 -*-
"""
asyncio-server.py
~~~~~~~~~~~~~~~~~

A fully-functional HTTP/2 server using asyncio. Requires Python 3.5+.

This example demonstrates handling requests with bodies, as well as handling
those without. In particular, it demonstrates the fact that DataReceived may
be called multiple times, and that applications must handle that possibility.
"""
import asyncio
import io
import json
```

(continues on next page)

(continued from previous page)

```
import ssl
import collections
from typing import List, Tuple

from h2.config import H2Configuration
from h2.connection import H2Connection
from h2.events import (
    ConnectionTerminated, DataReceived, RemoteSettingsChanged,
    RequestReceived, StreamEnded, StreamReset, WindowUpdated
)
from h2.errors import ErrorCodes
from h2.exceptions import ProtocolError, StreamClosedError
from h2.settings import SettingCodes


RequestData = collections.namedtuple('RequestData', ['headers', 'data'])


class H2Protocol(asyncio.Protocol):
    def __init__(self):
        config = H2Configuration(client_side=False, header_encoding='utf-8')
        self.conn = H2Connection(config=config)
        self.transport = None
        self.stream_data = {}
        self.flow_control_futures = {}

    def connection_made(self, transport: asyncio.Transport):
        self.transport = transport
        self.conn.initiate_connection()
        self.transport.write(self.conn.data_to_send())

    def connection_lost(self, exc):
        for future in self.flow_control_futures.values():
            future.cancel()
        self.flow_control_futures = {}

    def data_received(self, data: bytes):
        try:
            events = self.conn.receive_data(data)
        except ProtocolError as e:
            self.transport.write(self.conn.data_to_send())
            self.transport.close()
        else:
            self.transport.write(self.conn.data_to_send())
            for event in events:
                if isinstance(event, RequestReceived):
                    self.request_received(event.headers, event.stream_id)
                elif isinstance(event, DataReceived):
                    self.receive_data(event.data, event.stream_id)
                elif isinstance(event, StreamEnded):
                    self.stream_complete(event.stream_id)
                elif isinstance(event, ConnectionTerminated):
                    self.transport.close()
                elif isinstance(event, StreamReset):
                    self.stream_reset(event.stream_id)
                elif isinstance(event, WindowUpdated):
                    self.window_updated(event.stream_id, event.delta)
```

(continues on next page)

(continued from previous page)

```
                elif isinstance(event, RemoteSettingsChanged):
                    if SettingCodes.INITIAL_WINDOW_SIZE in event.changed_settings:
                        self.window_updated(None, 0)

                self.transport.write(self.conn.data_to_send())

    def request_received(self, headers: List[Tuple[str, str]], stream_id: int):
        headers = collections.OrderedDict(headers)
        method = headers[':method']

        # Store off the request data.
        request_data = RequestData(headers, io.BytesIO())
        self.stream_data[stream_id] = request_data

    def stream_complete(self, stream_id: int):
        """
        When a stream is complete, we can send our response.
        """
        try:
            request_data = self.stream_data[stream_id]
        except KeyError:
            # Just return, we probably 405'd this already
            return

        headers = request_data.headers
        body = request_data.data.getvalue().decode('utf-8')

        data = json.dumps(
            {"headers": headers, "body": body}, indent=4
        ).encode("utf8")

        response_headers = (
            (':status', '200'),
            ('content-type', 'application/json'),
            ('content-length', str(len(data))),
            ('server', 'asyncio-h2'),
        )
        self.conn.send_headers(stream_id, response_headers)
        asyncio.ensure_future(self.send_data(data, stream_id))

    def receive_data(self, data: bytes, stream_id: int):
        """
        We've received some data on a stream. If that stream is one we're
        expecting data on, save it off. Otherwise, reset the stream.
        """
        try:
            stream_data = self.stream_data[stream_id]
        except KeyError:
            self.conn.reset_stream(
                stream_id, error_code=ErrorCodes.PROTOCOL_ERROR
            )
        else:
            stream_data.data.write(data)

    def stream_reset(self, stream_id):
        """
        A stream reset was sent. Stop sending data.
```

(continues on next page)

(continued from previous page)

```
        """
        if stream_id in self.flow_control_futures:
            future = self.flow_control_futures.pop(stream_id)
            future.cancel()

    async def send_data(self, data, stream_id):
        """
        Send data according to the flow control rules.
        """
        while data:
            while self.conn.local_flow_control_window(stream_id) < 1:
                try:
                    await self.wait_for_flow_control(stream_id)
                except asyncio.CancelledError:
                    return

            chunk_size = min(
                self.conn.local_flow_control_window(stream_id),
                len(data),
                self.conn.max_outbound_frame_size,
            )

            try:
                self.conn.send_data(
                    stream_id,
                    data[:chunk_size],
                    end_stream=(chunk_size == len(data))
                )
            except (StreamClosedError, ProtocolError):
                # The stream got closed and we didn't get told. We're done
                # here.
                break

            self.transport.write(self.conn.data_to_send())
            data = data[chunk_size:]

    async def wait_for_flow_control(self, stream_id):
        """
        Waits for a Future that fires when the flow control window is opened.
        """
        f = asyncio.Future()
        self.flow_control_futures[stream_id] = f
        await f

    def window_updated(self, stream_id, delta):
        """
        A window update frame was received. Unblock some number of flow control
        Futures.
        """
        if stream_id and stream_id in self.flow_control_futures:
            f = self.flow_control_futures.pop(stream_id)
            f.set_result(delta)
        elif not stream_id:
            for f in self.flow_control_futures.values():
                f.set_result(delta)

            self.flow_control_futures = {}
```

(continues on next page)

(continued from previous page)

```


ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ssl_context.options |= (
    ssl.OP_NO_TLSv1 | ssl.OP_NO_TLSv1_1 | ssl.OP_NO_COMPRESSION
)
ssl_context.load_cert_chain(certfile="cert.crt", keyfile="cert.key")
ssl_context.set_alpn_protocols(["h2"])

loop = asyncio.get_event_loop()
# Each client connection will create a new protocol instance
coro = loop.create_server(H2Protocol, '127.0.0.1', 8443, ssl=ssl_context)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

Curio Example Server

This example is a basic HTTP/2 server written using curio, David Beazley's example of how to build a concurrent networking framework using Python 3.5's new async/await syntax.

This example is notable for demonstrating the correct use of HTTP/2 flow control with h2. It is also a good example of the brand new syntax.

```
#!/usr/bin/env python3.5
# -*- coding: utf-8 -*-
"""
curio-server.py
~~~~~~~~~~~~~~~

A fully-functional HTTP/2 server written for curio.

Requires Python 3.5+.
"""
import mimetypes
import os
import sys

from curio import Event, spawn, socket, ssl, run

import h2.config
import h2.connection
import h2.events


```

(continues on next page)

(continued from previous page)

```python
# The maximum amount of a file we'll send in a single DATA frame.
READ_CHUNK_SIZE = 8192


async def create_listening_ssl_socket(address, certfile, keyfile):
    """
    Create and return a listening TLS socket on a given address.
    """
    ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
    ssl_context.options |= (
        ssl.OP_NO_TLSv1 | ssl.OP_NO_TLSv1_1 | ssl.OP_NO_COMPRESSION
    )
    ssl_context.set_ciphers("ECDHE+AESGCM")
    ssl_context.load_cert_chain(certfile=certfile, keyfile=keyfile)
    ssl_context.set_alpn_protocols(["h2"])

    sock = socket.socket()
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock = await ssl_context.wrap_socket(sock)
    sock.bind(address)
    sock.listen()

    return sock


async def h2_server(address, root, certfile, keyfile):
    """
    Create an HTTP/2 server at the given address.
    """
    sock = await create_listening_ssl_socket(address, certfile, keyfile)
    print("Now listening on %s:%d" % address)

    async with sock:
        while True:
            client, _ = await sock.accept()
            server = H2Server(client, root)
            await spawn(server.run())


class H2Server:
    """
    A basic HTTP/2 file server. This is essentially very similar to
    SimpleHTTPServer from the standard library, but uses HTTP/2 instead of
    HTTP/1.1.
    """
    def __init__(self, sock, root):
        config = h2.config.H2Configuration(
            client_side=False, header_encoding='utf-8'
        )
        self.sock = sock
        self.conn = h2.connection.H2Connection(config=config)
        self.root = root
        self.flow_control_events = {}

    async def run(self):
        """
        Loop over the connection, managing it appropriately.
```

(continues on next page)

(continued from previous page)

```
        """
        self.conn.initiate_connection()
        await self.sock.sendall(self.conn.data_to_send())

        while True:
            # 65535 is basically arbitrary here: this amounts to "give me
            # whatever data you have".
            data = await self.sock.recv(65535)
            if not data:
                break

            events = self.conn.receive_data(data)
            for event in events:
                if isinstance(event, h2.events.RequestReceived):
                    await spawn(
                        self.request_received(event.headers, event.stream_id)
                    )
                elif isinstance(event, h2.events.DataReceived):
                    self.conn.reset_stream(event.stream_id)
                elif isinstance(event, h2.events.WindowUpdated):
                    await self.window_updated(event)

            await self.sock.sendall(self.conn.data_to_send())

    async def request_received(self, headers, stream_id):
        """
        Handle a request by attempting to serve a suitable file.
        """
        headers = dict(headers)
        assert headers[':method'] == 'GET'

        path = headers[':path'].lstrip('/')
        full_path = os.path.join(self.root, path)

        if not os.path.exists(full_path):
            response_headers = (
                (':status', '404'),
                ('content-length', '0'),
                ('server', 'curio-h2'),
            )
            self.conn.send_headers(
                stream_id, response_headers, end_stream=True
            )
            await self.sock.sendall(self.conn.data_to_send())
        else:
            await self.send_file(full_path, stream_id)

    async def send_file(self, file_path, stream_id):
        """
        Send a file, obeying the rules of HTTP/2 flow control.
        """
        filesize = os.stat(file_path).st_size
        content_type, content_encoding = mimetypes.guess_type(file_path)
        response_headers = [
            (':status', '200'),
            ('content-length', str(filesize)),
            ('server', 'curio-h2'),
```

(continues on next page)

(continued from previous page)

```
        ]
        if content_type:
            response_headers.append(('content-type', content_type))
        if content_encoding:
            response_headers.append(('content-encoding', content_encoding))

        self.conn.send_headers(stream_id, response_headers)
        await self.sock.sendall(self.conn.data_to_send())

        with open(file_path, 'rb', buffering=0) as f:
            await self._send_file_data(f, stream_id)

    async def _send_file_data(self, fileobj, stream_id):
        """
        Send the data portion of a file. Handles flow control rules.
        """
        while True:
            while self.conn.local_flow_control_window(stream_id) < 1:
                await self.wait_for_flow_control(stream_id)

            chunk_size = min(
                self.conn.local_flow_control_window(stream_id),
                READ_CHUNK_SIZE,
            )

            data = fileobj.read(chunk_size)
            keep_reading = (len(data) == chunk_size)

            self.conn.send_data(stream_id, data, not keep_reading)
            await self.sock.sendall(self.conn.data_to_send())

            if not keep_reading:
                break

    async def wait_for_flow_control(self, stream_id):
        """
        Blocks until the flow control window for a given stream is opened.
        """
        evt = Event()
        self.flow_control_events[stream_id] = evt
        await evt.wait()

    async def window_updated(self, event):
        """
        Unblock streams waiting on flow control, if needed.
        """
        stream_id = event.stream_id

        if stream_id and stream_id in self.flow_control_events:
            evt = self.flow_control_events.pop(stream_id)
            await evt.set()
        elif not stream_id:
            # Need to keep a real list here to use only the events present at
            # this time.
            blocked_streams = list(self.flow_control_events.keys())
            for stream_id in blocked_streams:
                event = self.flow_control_events.pop(stream_id)
```

(continues on next page)

(continued from previous page)

```
                await event.set()
        return


if __name__ == '__main__':
    host = sys.argv[2] if len(sys.argv) > 2 else "localhost"
    print("Try GETting:")
    print("    On OSX after 'brew install curl --with-c-ares --with-libidn --with-nghttp2 --with-openssl':")
    print("/usr/local/opt/curl/bin/curl --tlsv1.2 --http2 -k https://localhost:5000/bundle.js")
    print("Or open a browser to: https://localhost:5000/")
    print("    (Accept all the warnings)")
    run(h2_server((host, 5000), sys.argv[1],
                  "{}.crt.pem".format(host),
                  "{}.key".format(host)), with_monitor=True)
```

Eventlet Example Server

This example is a basic HTTP/2 server written using the eventlet concurrent networking framework. This example is notable for demonstrating how to configure PyOpenSSL, which eventlet uses for its TLS layer.

In terms of HTTP/2 functionality, this example is very simple: it returns the request headers as a JSON document to the caller. It does not obey HTTP/2 flow control, which is a flaw, but it is otherwise functional.

```
# -*- coding: utf-8 -*-
"""
eventlet-server.py
~~~~~~~~~~~~~~~~~~

A fully-functional HTTP/2 server written for Eventlet.
"""
import collections
import json

import eventlet

from eventlet.green.OpenSSL import SSL, crypto
from h2.config import H2Configuration
from h2.connection import H2Connection
from h2.events import RequestReceived, DataReceived


class ConnectionManager(object):
    """
    An object that manages a single HTTP/2 connection.
    """
    def __init__(self, sock):
        config = H2Configuration(client_side=False)
        self.sock = sock
        self.conn = H2Connection(config=config)

    def run_forever(self):
        self.conn.initiate_connection()
        self.sock.sendall(self.conn.data_to_send())
```

(continues on next page)

(continued from previous page)

```

        while True:
            data = self.sock.recv(65535)
            if not data:
                break

            events = self.conn.receive_data(data)

            for event in events:
                if isinstance(event, RequestReceived):
                    self.request_received(event.headers, event.stream_id)
                elif isinstance(event, DataReceived):
                    self.conn.reset_stream(event.stream_id)

            self.sock.sendall(self.conn.data_to_send())

    def request_received(self, headers, stream_id):
        headers = collections.OrderedDict(headers)
        data = json.dumps({'headers': headers}, indent=4).encode('utf-8')

        response_headers = (
            (':status', '200'),
            ('content-type', 'application/json'),
            ('content-length', str(len(data))),
            ('server', 'eventlet-h2'),
        )
        self.conn.send_headers(stream_id, response_headers)
        self.conn.send_data(stream_id, data, end_stream=True)


def alpn_callback(conn, protos):
    if b'h2' in protos:
        return b'h2'

    raise RuntimeError("No acceptable protocol offered!")


def npn_advertise_cb(conn):
    return [b'h2']


# Let's set up SSL. This is a lot of work in PyOpenSSL.
options = (
    SSL.OP_NO_COMPRESSION |
    SSL.OP_NO_SSLv2 |
    SSL.OP_NO_SSLv3 |
    SSL.OP_NO_TLSv1 |
    SSL.OP_NO_TLSv1_1
)
context = SSL.Context(SSL.SSLv23_METHOD)
context.set_options(options)
context.set_verify(SSL.VERIFY_NONE, lambda *args: True)
context.use_privatekey_file('server.key')
context.use_certificate_file('server.crt')
context.set_npn_advertise_callback(npn_advertise_cb)
context.set_alpn_select_callback(alpn_callback)
context.set_cipher_list(
```

(continues on next page)

(continued from previous page)

```
    "ECDHE+AESGCM"
)
context.set_tmp_ecdh(crypto.get_elliptic_curve(u'prime256v1'))

server = eventlet.listen(('0.0.0.0', 443))
server = SSL.Connection(context, server)
pool = eventlet.GreenPool()

while True:
    try:
        new_sock, _ = server.accept()
        manager = ConnectionManager(new_sock)
        pool.spawn_n(manager.run_forever)
    except (SystemExit, KeyboardInterrupt):
        break
```

Gevent Example Server

This example is a basic HTTP/2 server written using gevent, a powerful coroutine-based Python networking library that uses greenlet to provide a high-level synchronous API on top of the libev or libuv event loop.

This example is inspired by the curio one and also demonstrates the correct use of HTTP/2 flow control with h2 and how gevent can be simple to use.

```
# -*- coding: utf-8 -*-
"""
gevent-server.py
================

A simple HTTP/2 server written for gevent serving static files from a directory␣
↪specified as input.
If no directory is provided, the current directory will be used.
"""
import mimetypes
import sys
from functools import partial
from pathlib import Path
from typing import Tuple, Dict, Optional

from gevent import socket, ssl
from gevent.event import Event
from gevent.server import StreamServer
from h2 import events
from h2.config import H2Configuration
from h2.connection import H2Connection


def get_http2_tls_context() -> ssl.SSLContext:
    ctx = ssl.create_default_context(purpose=ssl.Purpose.CLIENT_AUTH)
    ctx.options |= (
            ssl.OP_NO_SSLv2 | ssl.OP_NO_SSLv3 | ssl.OP_NO_TLSv1 | ssl.OP_NO_TLSv1_1
    )

    ctx.options |= ssl.OP_NO_COMPRESSION
    ctx.set_ciphers('ECDHE+AESGCM:ECDHE+CHACHA20:DHE+AESGCM:DHE+CHACHA20')
```

(continues on next page)

(continued from previous page)

```
    ctx.load_cert_chain(certfile='localhost.crt', keyfile='localhost.key')
    ctx.set_alpn_protocols(['h2'])
    try:
        ctx.set_npn_protocols(['h2'])
    except NotImplementedError:
        pass

    return ctx


class H2Worker:

    def __init__(self, sock: socket, address: Tuple[str, str], source_dir: str = None):
        self._sock = sock
        self._address = address
        self._flow_control_events: Dict[int, Event] = {}
        self._server_name = 'gevent-h2'
        self._connection: Optional[H2Connection] = None
        self._read_chunk_size = 8192  # The maximum amount of a file we'll send in a single DATA frame

        self._check_sources_dir(source_dir)
        self._sources_dir = source_dir

        self._run()

    def _initiate_connection(self):
        config = H2Configuration(client_side=False, header_encoding='utf-8')
        self._connection = H2Connection(config=config)
        self._connection.initiate_connection()
        self._sock.sendall(self._connection.data_to_send())

    @staticmethod
    def _check_sources_dir(sources_dir: str) -> None:
        p = Path(sources_dir)
        if not p.is_dir():
            raise NotADirectoryError(f'{sources_dir} does not exists')

    def _send_error_response(self, status_code: str, event: events.RequestReceived) -> None:
        self._connection.send_headers(
            stream_id=event.stream_id,
            headers=[
                (':status', status_code),
                ('content-length', '0'),
                ('server', self._server_name),
            ],
            end_stream=True
        )
        self._sock.sendall(self._connection.data_to_send())

    def _handle_request(self, event: events.RequestReceived) -> None:
        headers = dict(event.headers)
        if headers[':method'] != 'GET':
            self._send_error_response('405', event)
            return
```

(continues on next page)

(continued from previous page)

```

        file_path = Path(self._sources_dir) / headers[':path'].lstrip('/')
        if not file_path.is_file():
            self._send_error_response('404', event)
            return

        self._send_file(file_path, event.stream_id)

    def _send_file(self, file_path: Path, stream_id: int) -> None:
        """
        Send a file, obeying the rules of HTTP/2 flow control.
        """
        file_size = file_path.stat().st_size
        content_type, content_encoding = mimetypes.guess_type(str(file_path))
        response_headers = [
            (':status', '200'),
            ('content-length', str(file_size)),
            ('server', self._server_name)
        ]
        if content_type:
            response_headers.append(('content-type', content_type))
        if content_encoding:
            response_headers.append(('content-encoding', content_encoding))

        self._connection.send_headers(stream_id, response_headers)
        self._sock.sendall(self._connection.data_to_send())

        with file_path.open(mode='rb', buffering=0) as f:
            self._send_file_data(f, stream_id)

    def _send_file_data(self, file_obj, stream_id: int) -> None:
        """
        Send the data portion of a file. Handles flow control rules.
        """
        while True:
            while self._connection.local_flow_control_window(stream_id) < 1:
                self._wait_for_flow_control(stream_id)

            chunk_size = min(self._connection.local_flow_control_window(stream_id),␣
↪self._read_chunk_size)
            data = file_obj.read(chunk_size)
            keep_reading = (len(data) == chunk_size)

            self._connection.send_data(stream_id, data, not keep_reading)
            self._sock.sendall(self._connection.data_to_send())

            if not keep_reading:
                break

    def _wait_for_flow_control(self, stream_id: int) -> None:
        """
        Blocks until the flow control window for a given stream is opened.
        """
        event = Event()
        self._flow_control_events[stream_id] = event
        event.wait()

```

(continues on next page)

(continued from previous page)

```
    def _handle_window_update(self, event: events.WindowUpdated) -> None:
        """
        Unblock streams waiting on flow control, if needed.
        """
        stream_id = event.stream_id

        if stream_id and stream_id in self._flow_control_events:
            g_event = self._flow_control_events.pop(stream_id)
            g_event.set()
        elif not stream_id:
            # Need to keep a real list here to use only the events present at this time.
            blocked_streams = list(self._flow_control_events.keys())
            for stream_id in blocked_streams:
                g_event = self._flow_control_events.pop(stream_id)
                g_event.set()

    def _run(self) -> None:
        self._initiate_connection()

        while True:
            data = self._sock.recv(65535)
            if not data:
                break

            h2_events = self._connection.receive_data(data)
            for event in h2_events:
                if isinstance(event, events.RequestReceived):
                    self._handle_request(event)
                elif isinstance(event, events.DataReceived):
                    self._connection.reset_stream(event.stream_id)
                elif isinstance(event, events.WindowUpdated):
                    self._handle_window_update(event)

            data_to_send = self._connection.data_to_send()
            if data_to_send:
                self._sock.sendall(data_to_send)


if __name__ == '__main__':
    files_dir = sys.argv[1] if len(sys.argv) > 1 else f'{Path().cwd()}'
    server = StreamServer(('127.0.0.1', 8080), partial(H2Worker, source_dir=files_dir),
                          ssl_context=get_http2_tls_context())
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        server.close()
```

Tornado Example Server

This example is a basic HTTP/2 server written using the Tornado asynchronous networking library.

The server returns the request headers as a JSON document to the caller, just like the example from the *Getting Started: Writing Your Own HTTP/2 Server* document.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
tornado-server.py
~~~~~~~~~~~~~~~~~

A fully-functional HTTP/2 server written for Tornado.
"""
import collections
import json
import ssl

import tornado.gen
import tornado.ioloop
import tornado.iostream
import tornado.tcpserver

from h2.config import H2Configuration
from h2.connection import H2Connection
from h2.events import RequestReceived, DataReceived


def create_ssl_context(certfile, keyfile):
    ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
    ssl_context.options |= (
        ssl.OP_NO_TLSv1 | ssl.OP_NO_TLSv1_1 | ssl.OP_NO_COMPRESSION
    )
    ssl_context.set_ciphers("ECDHE+AESGCM")
    ssl_context.load_cert_chain(certfile=certfile, keyfile=keyfile)
    ssl_context.set_alpn_protocols(["h2"])
    return ssl_context


class H2Server(tornado.tcpserver.TCPServer):

    @tornado.gen.coroutine
    def handle_stream(self, stream, address):
        handler = EchoHeadersHandler(stream)
        yield handler.handle()


class EchoHeadersHandler(object):

    def __init__(self, stream):
        self.stream = stream

        config = H2Configuration(client_side=False)
        self.conn = H2Connection(config=config)

    @tornado.gen.coroutine
    def handle(self):
        self.conn.initiate_connection()
        yield self.stream.write(self.conn.data_to_send())

        while True:
            try:
                data = yield self.stream.read_bytes(65535, partial=True)
```

(continues on next page)

(continued from previous page)

```
                if not data:
                    break

                events = self.conn.receive_data(data)
                for event in events:
                    if isinstance(event, RequestReceived):
                        self.request_received(event.headers, event.stream_id)
                    elif isinstance(event, DataReceived):
                        self.conn.reset_stream(event.stream_id)

                yield self.stream.write(self.conn.data_to_send())

            except tornado.iostream.StreamClosedError:
                break

    def request_received(self, headers, stream_id):
        headers = collections.OrderedDict(headers)
        data = json.dumps({'headers': headers}, indent=4).encode('utf-8')

        response_headers = (
            (':status', '200'),
            ('content-type', 'application/json'),
            ('content-length', str(len(data))),
            ('server', 'tornado-h2'),
        )
        self.conn.send_headers(stream_id, response_headers)
        self.conn.send_data(stream_id, data, end_stream=True)


if __name__ == '__main__':
    ssl_context = create_ssl_context('server.crt', 'server.key')
    server = H2Server(ssl_options=ssl_context)
    server.listen(8888)
    io_loop = tornado.ioloop.IOLoop.current()
    io_loop.start()
```

Twisted Example Server

This example is a basic HTTP/2 server written for the Twisted asynchronous networking framework. This is a relatively fleshed out example, and in particular it makes sure to obey HTTP/2 flow control rules.

This server differs from some of the other example servers by serving files, rather than simply sending JSON responses. This makes the example lengthier, but also brings it closer to a real-world use-case.

```
# -*- coding: utf-8 -*-
"""
twisted-server.py
~~~~~~~~~~~~~~~~~

A fully-functional HTTP/2 server written for Twisted.
"""
import functools
import mimetypes
import os
import os.path
```

(continues on next page)

(continued from previous page)

```python
import sys

from OpenSSL import crypto
from twisted.internet.defer import Deferred, inlineCallbacks
from twisted.internet.protocol import Protocol, Factory
from twisted.internet import endpoints, reactor, ssl
from h2.config import H2Configuration
from h2.connection import H2Connection
from h2.events import (
    RequestReceived, DataReceived, WindowUpdated
)
from h2.exceptions import ProtocolError


def close_file(file, d):
    file.close()


READ_CHUNK_SIZE = 8192


class H2Protocol(Protocol):
    def __init__(self, root):
        config = H2Configuration(client_side=False)
        self.conn = H2Connection(config=config)
        self.known_proto = None
        self.root = root

        self._flow_control_deferreds = {}

    def connectionMade(self):
        self.conn.initiate_connection()
        self.transport.write(self.conn.data_to_send())

    def dataReceived(self, data):
        if not self.known_proto:
            self.known_proto = True

        try:
            events = self.conn.receive_data(data)
        except ProtocolError:
            if self.conn.data_to_send:
                self.transport.write(self.conn.data_to_send())
            self.transport.loseConnection()
        else:
            for event in events:
                if isinstance(event, RequestReceived):
                    self.requestReceived(event.headers, event.stream_id)
                elif isinstance(event, DataReceived):
                    self.dataFrameReceived(event.stream_id)
                elif isinstance(event, WindowUpdated):
                    self.windowUpdated(event)

            if self.conn.data_to_send:
                self.transport.write(self.conn.data_to_send())

    def requestReceived(self, headers, stream_id):
```

(continues on next page)

(continued from previous page)

```
        headers = dict(headers)  # Invalid conversion, fix later.
        assert headers[b':method'] == b'GET'

        path = headers[b':path'].lstrip(b'/')
        full_path = os.path.join(self.root, path)

        if not os.path.exists(full_path):
            response_headers = (
                (':status', '404'),
                ('content-length', '0'),
                ('server', 'twisted-h2'),
            )
            self.conn.send_headers(
                stream_id, response_headers, end_stream=True
            )
            self.transport.write(self.conn.data_to_send())
        else:
            self.sendFile(full_path, stream_id)

        return

    def dataFrameReceived(self, stream_id):
        self.conn.reset_stream(stream_id)
        self.transport.write(self.conn.data_to_send())

    def sendFile(self, file_path, stream_id):
        filesize = os.stat(file_path).st_size
        content_type, content_encoding = mimetypes.guess_type(
            file_path.decode('utf-8')
        )
        response_headers = [
            (':status', '200'),
            ('content-length', str(filesize)),
            ('server', 'twisted-h2'),
        ]
        if content_type:
            response_headers.append(('content-type', content_type))
        if content_encoding:
            response_headers.append(('content-encoding', content_encoding))

        self.conn.send_headers(stream_id, response_headers)
        self.transport.write(self.conn.data_to_send())

        f = open(file_path, 'rb')
        d = self._send_file(f, stream_id)
        d.addErrback(functools.partial(close_file, f))

    def windowUpdated(self, event):
        """
        Handle a WindowUpdated event by firing any waiting data sending
        callbacks.
        """
        stream_id = event.stream_id

        if stream_id and stream_id in self._flow_control_deferreds:
            d = self._flow_control_deferreds.pop(stream_id)
            d.callback(event.delta)
```

(continues on next page)

(continued from previous page)

```
        elif not stream_id:
            for d in self._flow_control_deferreds.values():
                d.callback(event.delta)

            self._flow_control_deferreds = {}

        return

    @inlineCallbacks
    def _send_file(self, file, stream_id):
        """
        This callback sends more data for a given file on the stream.
        """
        keep_reading = True
        while keep_reading:
            while not self.conn.remote_flow_control_window(stream_id):
                yield self.wait_for_flow_control(stream_id)

            chunk_size = min(
                self.conn.remote_flow_control_window(stream_id), READ_CHUNK_SIZE
            )
            data = file.read(chunk_size)
            keep_reading = len(data) == chunk_size
            self.conn.send_data(stream_id, data, not keep_reading)
            self.transport.write(self.conn.data_to_send())

            if not keep_reading:
                break

        file.close()

    def wait_for_flow_control(self, stream_id):
        """
        Returns a Deferred that fires when the flow control window is opened.
        """
        d = Deferred()
        self._flow_control_deferreds[stream_id] = d
        return d


class H2Factory(Factory):
    def __init__(self, root):
        self.root = root

    def buildProtocol(self, addr):
        print(H2Protocol)
        return H2Protocol(self.root)


root = sys.argv[1].encode('utf-8')

with open('server.crt', 'r') as f:
    cert_data = f.read()
with open('server.key', 'r') as f:
    key_data = f.read()

cert = crypto.load_certificate(crypto.FILETYPE_PEM, cert_data)
```

(continues on next page)

(continued from previous page)

```
key = crypto.load_privatekey(crypto.FILETYPE_PEM, key_data)
options = ssl.CertificateOptions(
    privateKey=key,
    certificate=cert,
    acceptableProtocols=[b'h2'],
)

endpoint = endpoints.SSL4ServerEndpoint(reactor, 8080, options, backlog=128)
endpoint.listen(H2Factory(root))
reactor.run()
```

Example HTTP/2-only WSGI Server

This example is a more complex HTTP/2 server that acts as a WSGI server, passing data to an arbitrary WSGI application. This example is written using asyncio. The server supports most of PEP-3333, and so could in principle be used as a production WSGI server: however, that's *not recommended* as certain shortcuts have been taken to ensure ease of implementation and understanding.

The main advantages of this example are:

1. It properly demonstrates HTTP/2 flow control management.
2. It demonstrates how to plug h2 into a larger, more complex application.

```
# -*- coding: utf-8 -*-
"""
asyncio-server.py
~~~~~~~~~~~~~~~~~

A fully-functional WSGI server, written using h2. Requires asyncio.

To test it, try installing httpbin from pip (``pip install httpbin``) and then
running the server (``python asyncio-server.py httpbin:app``).

This server does not support HTTP/1.1: it is a HTTP/2-only WSGI server. The
purpose of this code is to demonstrate how to integrate h2 into a more
complex application, and to demonstrate several principles of concurrent
programming.

The architecture looks like this:

+---------------------------------+
|     1x HTTP/2 Server Thread     |
|        (running asyncio)        |
+---------------------------------+
+---------------------------------+
|    N WSGI Application Threads   |
|           (no asyncio)          |
+---------------------------------+

Essentially, we spin up an asyncio-based event loop in the main thread. This
launches one HTTP/2 Protocol instance for each inbound connection, all of which
will read and write data from within the main thread in an asynchronous manner.

When each HTTP request comes in, the server will build the WSGI environment
dictionary and create a ``Stream`` object. This object will hold the relevant
```

(continues on next page)

(continued from previous page)

```
state for the request/response pair and will act as the WSGI side of the logic.
That object will then be passed to a background thread pool, and when a worker
is available the WSGI logic will begin to be executed. This model ensures that
the asyncio web server itself is never blocked by the WSGI application.

The WSGI application and the HTTP/2 server communicate via an asyncio queue,
together with locks and threading events. The locks themselves are implicit in
asyncio's "call_soon_threadsafe", which allows for a background thread to
register an action with the main asyncio thread. When the asyncio thread
eventually takes the action in question it sets as threading event, signaling
to the background thread that it is free to continue its work.

To make the WSGI application work with flow control, there is a very important
invariant that must be observed. Any WSGI action that would cause data to be
emitted to the network MUST be accompanied by a threading Event that is not
set until that data has been written to the transport. This ensures that the
WSGI application *blocks* until the data is actually sent. The reason we
require this invariant is that the HTTP/2 server may choose to re-order some
data chunks for flow control reasons: that is, the application for stream X may
have actually written its data first, but the server may elect to send the data
for stream Y first. This means that it's vital that there not be *two* writes
for stream X active at any one point or they may get reordered, which would be
particularly terrible.

Thus, the server must cooperate to ensure that each threading event only fires
when the *complete* data for that event has been written to the asyncio
transport. Any earlier will cause untold craziness.
"""
import asyncio
import importlib
import queue
import ssl
import sys
import threading

from h2.config import H2Configuration
from h2.connection import H2Connection
from h2.events import (
    DataReceived, RequestReceived, WindowUpdated, StreamEnded, StreamReset
)


# Used to signal that a request has completed.
#
# This is a convenient way to do "in-band" signaling of stream completion
# without doing anything so heavyweight as using a class. Essentially, we can
# test identity against this empty object. In fact, this is so convenient that
# we use this object for all streams, for data in both directions: in and out.
END_DATA_SENTINEL = object()

# The WSGI callable. Stored here so that the protocol instances can get hold
# of the data.
APPLICATION = None


class H2Protocol(asyncio.Protocol):
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        config = H2Configuration(client_side=False, header_encoding='utf-8')

        # Our server-side state machine.
        self.conn = H2Connection(config=config)

        # The backing transport.
        self.transport = None

        # A dictionary of ``Stream`` objects, keyed by their stream ID. This
        # makes it easy to route data to the correct WSGI application instance.
        self.streams = {}

        # A queue of data emitted by WSGI applications that has not yet been
        # sent. Each stream may only have one chunk of data in either this
        # queue or the flow_controlled_data dictionary at any one time.
        self._stream_data = asyncio.Queue()

        # Data that has been pulled off the queue that is for a stream blocked
        # behind flow control limitations. This is used to avoid spinning on
        # _stream_data queue when a stream cannot have its data sent. Data that
        # cannot be sent on the connection when it is popped off the queue gets
        # placed here until the stream flow control window opens up again.
        self._flow_controlled_data = {}

        # A reference to the loop in which this protocol runs. This is needed
        # to synchronise up with background threads.
        self._loop = asyncio.get_event_loop()

        # Any streams that have been remotely reset. We keep track of these to
        # ensure that we don't emit data from a WSGI application whose stream
        # has been cancelled.
        self._reset_streams = set()

        # Keep track of the loop sending task so we can kill it when the
        # connection goes away.
        self._send_loop_task = None

    def connection_made(self, transport):
        """
        The connection has been made. Here we need to save off our transport,
        do basic HTTP/2 connection setup, and then start our data writing
        coroutine.
        """
        self.transport = transport
        self.conn.initiate_connection()
        self.transport.write(self.conn.data_to_send())
        self._send_loop_task = self._loop.create_task(self.sending_loop())

    def connection_lost(self, exc):
        """
        With the end of the connection, we just want to cancel our data sending
        coroutine.
        """
        self._send_loop_task.cancel()

    def data_received(self, data):
        """
```

(continues on next page)

(continued from previous page)

```
        Process inbound data.
        """
        events = self.conn.receive_data(data)

        for event in events:
            if isinstance(event, RequestReceived):
                self.request_received(event)
            elif isinstance(event, DataReceived):
                self.data_frame_received(event)
            elif isinstance(event, WindowUpdated):
                self.window_opened(event)
            elif isinstance(event, StreamEnded):
                self.end_stream(event)
            elif isinstance(event, StreamReset):
                self.reset_stream(event)

        outbound_data = self.conn.data_to_send()
        if outbound_data:
            self.transport.write(outbound_data)

    def window_opened(self, event):
        """
        The flow control window got opened.

        This is important because it's possible that we were unable to send
        some WSGI data because the flow control window was too small. If that
        happens, the sending_loop coroutine starts buffering data.

        As the window gets opened, we need to unbuffer the data. We do that by
        placing the data chunks back on the back of the send queue and letting
        the sending loop take another shot at sending them.

        This system only works because we require that each stream only have
        *one* data chunk in the sending queue at any time. The threading events
        force this invariant to remain true.
        """
        if event.stream_id:
            # This is specific to a single stream.
            if event.stream_id in self._flow_controlled_data:
                self._stream_data.put_nowait(
                    self._flow_controlled_data.pop(event.stream_id)
                )
        else:
            # This event is specific to the connection. Free up *all* the
            # streams. This is a bit tricky, but we *must not* yield the flow
            # of control here or it all goes wrong.
            for data in self._flow_controlled_data.values():
                self._stream_data.put_nowait(data)

            self._flow_controlled_data = {}

    @asyncio.coroutine
    def sending_loop(self):
        """
        A call that loops forever, attempting to send data. This sending loop
        contains most of the flow-control smarts of this class: it pulls data
        off of the asyncio queue and then attempts to send it.
```

(continues on next page)

(continued from previous page)

```

        The difficulties here are all around flow control. Specifically, a
        chunk of data may be too large to send. In this case, what will happen
        is that this coroutine will attempt to send what it can and will then
        store the unsent data locally. When a flow control event comes in that
        data will be freed up and placed back onto the asyncio queue, causing
        it to pop back up into the sending logic of this coroutine.

        This method explicitly *does not* handle HTTP/2 priority. That adds an
        extra layer of complexity to what is already a fairly complex method,
        and we'll look at how to do it another time.

        This coroutine explicitly *does not end*.
        """
        while True:
            stream_id, data, event = yield from self._stream_data.get()

            # If this stream got reset, just drop the data on the floor. Note
            # that we need to reset the event here to make sure that
            # application doesn't lock up.
            if stream_id in self._reset_streams:
                event.set()

            # Check if the body is done. If it is, this is really easy! Again,
            # we *must* set the event here or the application will lock up.
            if data is END_DATA_SENTINEL:
                self.conn.end_stream(stream_id)
                self.transport.write(self.conn.data_to_send())
                event.set()
                continue

            # We need to send data, but not to exceed the flow control window.
            # For that reason, grab only the data that fits: we'll buffer the
            # rest.
            window_size = self.conn.local_flow_control_window(stream_id)
            chunk_size = min(window_size, len(data))
            data_to_send = data[:chunk_size]
            data_to_buffer = data[chunk_size:]

            if data_to_send:
                # There's a maximum frame size we have to respect. Because we
                # aren't paying any attention to priority here, we can quite
                # safely just split this string up into chunks of max frame
                # size and blast them out.
                #
                # In a *real* application you'd want to consider priority here.
                max_size = self.conn.max_outbound_frame_size
                chunks = (
                    data_to_send[x:x+max_size]
                    for x in range(0, len(data_to_send), max_size)
                )
                for chunk in chunks:
                    self.conn.send_data(stream_id, chunk)
                self.transport.write(self.conn.data_to_send())

            # If there's data left to buffer, we should do that. Put it in a
            # dictionary and *don't set the event*: the app must not generate
```

(continues on next page)

(continued from previous page)

```
            # any more data until we got rid of all of this data.
            if data_to_buffer:
                self._flow_controlled_data[stream_id] = (
                    stream_id, data_to_buffer, event
                )
            else:
                # We sent everything. We can let the WSGI app progress.
                event.set()

    def request_received(self, event):
        """
        A HTTP/2 request has been received. We need to invoke the WSGI
        application in a background thread to handle it.
        """
        # First, we are going to want an object to hold all the relevant state
        # for this request/response. For that, we have a stream object. We
        # need to store the stream object somewhere reachable for when data
        # arrives later.
        s = Stream(event.stream_id, self)
        self.streams[event.stream_id] = s

        # Next, we need to build the WSGI environ dictionary.
        environ = _build_environ_dict(event.headers, s)

        # Finally, we want to throw these arguments out to a threadpool and
        # let it run.
        self._loop.run_in_executor(
            None,
            s.run_in_threadpool,
            APPLICATION,
            environ,
        )

    def data_frame_received(self, event):
        """
        Data has been received by WSGI server and needs to be dispatched to a
        running application.

        Note that the flow control window is not modified here. That's
        deliberate: see Stream.__next__ for a longer discussion of why.
        """
        # Grab the stream in question from our dictionary and pass it on.
        stream = self.streams[event.stream_id]
        stream.receive_data(event.data, event.flow_controlled_length)

    def end_stream(self, event):
        """
        The stream data is complete.
        """
        stream = self.streams[event.stream_id]
        stream.request_complete()

    def reset_stream(self, event):
        """
        A stream got forcefully reset.

        This is a tricky thing to deal with because WSGI doesn't really have a
```

(continues on next page)

(continued from previous page)

```
        good notion for it. Essentially, you have to let the application run
        until completion, but not actually let it send any data.

        We do that by discarding any data we currently have for it, and then
        marking the stream as reset to allow us to spot when that stream is
        trying to send data and drop that data on the floor.

        We then *also* signal the WSGI application that no more data is
        incoming, to ensure that it does not attempt to do further reads of the
        data.
        """
        if event.stream_id in self._flow_controlled_data:
            del self._flow_controlled_data

        self._reset_streams.add(event.stream_id)
        self.end_stream(event)

    def data_for_stream(self, stream_id, data):
        """
        Thread-safe method called from outside the main asyncio thread in order
        to send data on behalf of a WSGI application.

        Places data being written by a stream on an asyncio queue. Returns a
        threading event that will fire when that data is sent.
        """
        event = threading.Event()
        self._loop.call_soon_threadsafe(
            self._stream_data.put_nowait,
            (stream_id, data, event)
        )
        return event

    def send_response(self, stream_id, headers):
        """
        Thread-safe method called from outside the main asyncio thread in order
        to send the HTTP response headers on behalf of a WSGI application.

        Returns a threading event that will fire when the headers have been
        emitted to the network.
        """
        event = threading.Event()

        def _inner_send(stream_id, headers, event):
            self.conn.send_headers(stream_id, headers, end_stream=False)
            self.transport.write(self.conn.data_to_send())
            event.set()

        self._loop.call_soon_threadsafe(
            _inner_send,
            stream_id,
            headers,
            event
        )
        return event

    def open_flow_control_window(self, stream_id, increment):
        """
```

(continues on next page)

(continued from previous page)

```
        Opens a flow control window for the given stream by the given amount.
        Called from a WSGI thread. Does not return an event because there's no
        need to block on this action, it may take place at any time.
        """
        def _inner_open(stream_id, increment):
            self.conn.increment_flow_control_window(increment, stream_id)
            self.conn.increment_flow_control_window(increment, None)
            self.transport.write(self.conn.data_to_send())

        self._loop.call_soon_threadsafe(
            _inner_open,
            stream_id,
            increment,
        )


class Stream:
    """
    This class holds all of the state for a single stream. It also provides
    several of the callables used by the WSGI application. Finally, it provides
    the logic for actually interfacing with the WSGI application.

    For these reasons, the object has *strict* requirements on thread-safety.
    While the object can be initialized in the main WSGI thread, the
    ``run_in_threadpool`` method *must* be called from outside that thread. At
    that point, the main WSGI thread may only call specific methods.
    """
    def __init__(self, stream_id, protocol):
        self.stream_id = stream_id
        self._protocol = protocol

        # Queue for data that has been received from the network. This is a
        # thread-safe queue, to allow both the WSGI application to block on
        # receiving more data and to allow the asyncio server to keep sending
        # more data.
        #
        # This queue is unbounded in size, but in practice it cannot contain
        # too much data because the flow control window doesn't get adjusted
        # unless data is removed from it.
        self._received_data = queue.Queue()

        # This buffer is used to hold partial chunks of data from
        # _received_data that were not returned out of ``read`` and friends.
        self._temp_buffer = b''

        # Temporary variables that allow us to keep hold of the headers and
        # response status until such time as the application needs us to send
        # them.
        self._response_status = b''
        self._response_headers = []
        self._headers_emitted = False

        # Whether the application has received all the data from the network
        # or not. This allows us to short-circuit some reads.
        self._complete = False

    def receive_data(self, data, flow_controlled_size):
```

(continues on next page)

(continued from previous page)

```
        """
        Called by the H2Protocol when more data has been received from the
        network.

        Places the data directly on the queue in a thread-safe manner without
        blocking. Does not introspect or process the data.
        """
        self._received_data.put_nowait((data, flow_controlled_size))

    def request_complete(self):
        """
        Called by the H2Protocol when all the request data has been received.

        This works by placing the ``END_DATA_SENTINEL`` on the queue. The
        reading code knows, when it sees the ``END_DATA_SENTINEL``, to expect
        no more data from the network. This ensures that the state of the
        application only changes when it has finished processing the data from
        the network, even though the server may have long-since finished
        receiving all the data for this request.
        """
        self._received_data.put_nowait((END_DATA_SENTINEL, None))

    def run_in_threadpool(self, wsgi_application, environ):
        """
        This method should be invoked in a threadpool. At the point this method
        is invoked, the only safe methods to call from the original thread are
        ``receive_data`` and ``request_complete``: any other method is unsafe.

        This method handles the WSGI logic. It invokes the application callable
        in this thread, passing control over to the WSGI application. It then
        ensures that the data makes it back to the HTTP/2 connection via
        the thread-safe APIs provided below.
        """
        result = wsgi_application(environ, self.start_response)

        try:
            for data in result:
                self.write(data)
        finally:
            # This signals that we're done with data. The server will know that
            # this allows it to clean up its state: we're done here.
            self.write(END_DATA_SENTINEL)

    # The next few methods are called by the WSGI application. Firstly, the
    # three methods provided by the input stream.
    def read(self, size=None):
        """
        Called by the WSGI application to read data.

        This method is the one of two that explicitly pumps the input data
        queue, which means it deals with the ``_complete`` flag and the
        ``END_DATA_SENTINEL``.
        """
        # If we've already seen the END_DATA_SENTINEL, return immediately.
        if self._complete:
            return b''

```

(continues on next page)

(continued from previous page)

```
        # If we've been asked to read everything, just iterate over ourselves.
        if size is None:
            return b''.join(self)

        # Otherwise, as long as we don't have enough data, spin looking for
        # another data chunk.
        data = b''
        while len(data) < size:
            try:
                chunk = next(self)
            except StopIteration:
                break

            # Concatenating strings this way is slow, but that's ok, this is
            # just a demo.
            data += chunk

        # We have *at least* enough data to return, but we may have too much.
        # If we do, throw it on a buffer: we'll use it later.
        to_return = data[:size]
        self._temp_buffer = data[size:]
        return to_return

    def readline(self, hint=None):
        """
        Called by the WSGI application to read a single line of data.

        This method rigorously observes the ``hint`` parameter: it will only
        ever read that much data. It then splits the data on a newline
        character and throws everything it doesn't need into a buffer.
        """
        data = self.read(hint)
        first_newline = data.find(b'\n')
        if first_newline == -1:
            # No newline, return all the data
            return data

        # We want to slice the data so that the head *includes* the first
        # newline. Then, any data left in this line we don't care about should
        # be prepended to the internal buffer.
        head, tail = data[:first_newline + 1], data[first_newline + 1:]
        self._temp_buffer = tail + self._temp_buffer

        return head

    def readlines(self, hint=None):
        """
        Called by the WSGI application to read several lines of data.

        This method is really pretty stupid. It rigorously observes the
        ``hint`` parameter, and quite happily returns the input split into
        lines.
        """
        # This method is *crazy inefficient*, but it's also a pretty stupid
        # method to call.
        data = self.read(hint)
        lines = data.split(b'\n')
```

(continues on next page)

(continued from previous page)

~~~python

        # Split removes the newline character, but we want it, so put it back.
        lines = [line + b'\n' for line in lines]

        # Except if the last character was a newline character we now have an
        # extra line that is just a newline: pull that out.
        if lines[-1] == b'\n':
            lines = lines[:-1]
        return lines

    def start_response(self, status, response_headers, exc_info=None):
        """
        This is the PEP-3333 mandated start_response callable.

        All it does is store the headers for later sending, and return our
        ```write`` callable.
        """
        if self._headers_emitted and exc_info is not None:
            raise exc_info[1].with_traceback(exc_info[2])

        assert not self._response_status or exc_info is not None
        self._response_status = status
        self._response_headers = response_headers

        return self.write

    def write(self, data):
        """
        Provides some data to write.

        This function *blocks* until such time as the data is allowed by
        HTTP/2 flow control. This allows a client to slow or pause the response
        as needed.

        This function is not supposed to be used, according to PEP-3333, but
        once we have it it becomes quite convenient to use it, so this app
        actually runs all writes through this function.
        """
        if not self._headers_emitted:
            self._emit_headers()
        event = self._protocol.data_for_stream(self.stream_id, data)
        event.wait()
        return

    def _emit_headers(self):
        """
        Sends the response headers.

        This is only called from the write callable and should only ever be
        called once. It does some minor processing (converts the status line
        into a status code because reason phrases are evil) and then passes
        the headers on to the server. This call explicitly blocks until the
        server notifies us that the headers have reached the network.
        """
        assert self._response_status and self._response_headers
        assert not self._headers_emitted
        self._headers_emitted = True
~~~

(continues on next page)

(continued from previous page)

```

        # We only need the status code
        status = self._response_status.split(" ", 1)[0]
        headers = [(":status", status)]
        headers.extend(self._response_headers)
        event = self._protocol.send_response(self.stream_id, headers)
        event.wait()
        return

    # These two methods implement the iterator protocol. This allows a WSGI
    # application to iterate over this Stream object to get the data.
    def __iter__(self):
        return self

    def __next__(self):
        # If the complete request has been read, abort immediately.
        if self._complete:
            raise StopIteration()

        # If we have data stored in a temporary buffer for any reason, return
        # that and clear the buffer.
        #
        # This can actually only happen when the application uses one of the
        # read* callables, but that's fine.
        if self._temp_buffer:
            buffered_data = self._temp_buffer
            self._temp_buffer = b''
            return buffered_data

        # Otherwise, pull data off the queue (blocking as needed). If this is
        # the end of the request, we're done here: mark ourselves as complete
        # and call it time. Otherwise, open the flow control window an
        # appropriate amount and hand the chunk off.
        chunk, chunk_size = self._received_data.get()
        if chunk is END_DATA_SENTINEL:
            self._complete = True
            raise StopIteration()

        # Let's talk a little bit about why we're opening the flow control
        # window *here*, and not in the server thread.
        #
        # The purpose of HTTP/2 flow control is to allow for servers and
        # clients to avoid needing to buffer data indefinitely because their
        # peer is producing data faster than they can consume it. As a result,
        # it's important that the flow control window be opened as late in the
        # processing as possible. In this case, we open the flow control window
        # exactly when the server hands the data to the application. This means
        # that the flow control window essentially signals to the remote peer
        # how much data hasn't even been *seen* by the application yet.
        #
        # If you wanted to be really clever you could consider not opening the
        # flow control window until the application asks for the *next* chunk
        # of data. That means that any buffers at the application level are now
        # included in the flow control window processing. In my opinion, the
        # advantage of that process does not outweigh the extra logical
        # complexity involved in doing it, so we don't bother here.
        #
```

(continues on next page)

(continued from previous page)

```
        # Another note: you'll notice that we don't include the _temp_buffer in
        # our flow control considerations. This means you could in principle
        # lead us to buffer slightly more than one connection flow control
        # window's worth of data. That risk is considered acceptable for the
        # much simpler logic available here.
        #
        # Finally, this is a pretty dumb flow control window management scheme:
        # it causes us to emit a *lot* of window updates. A smarter server
        # would want to use the content-length header to determine whether
        # flow control window updates need to be emitted at all, and then to be
        # more efficient about emitting them to avoid firing them off really
        # frequently. For an example like this, there's very little gained by
        # worrying about that.
        self._protocol.open_flow_control_window(self.stream_id, chunk_size)

        return chunk


def _build_environ_dict(headers, stream):
    """
    Build the WSGI environ dictionary for a given request. To do that, we'll
    temporarily create a dictionary for the headers. While this isn't actually
    a valid way to represent headers, we know that the special headers we need
    can only have one appearance in the block.

    This code is arguably somewhat incautious: the conversion to dictionary
    should only happen in a way that allows us to correctly join headers that
    appear multiple times. That's acceptable in a demo app: in a productised
    version you'd want to fix it.
    """
    header_dict = dict(headers)
    path = header_dict.pop(u':path')
    try:
        path, query = path.split(u'?', 1)
    except ValueError:
        query = u""
    server_name = header_dict.pop(u':authority')
    try:
        server_name, port = server_name.split(u':', 1)
    except ValueError as e:
        port = "8443"

    environ = {
        u'REQUEST_METHOD': header_dict.pop(u':method'),
        u'SCRIPT_NAME': u'',
        u'PATH_INFO': path,
        u'QUERY_STRING': query,
        u'SERVER_NAME': server_name,
        u'SERVER_PORT': port,
        u'SERVER_PROTOCOL': u'HTTP/2',
        u'HTTPS': u"on",
        u'SSL_PROTOCOL': u'TLSv1.2',
        u'wsgi.version': (1, 0),
        u'wsgi.url_scheme': header_dict.pop(u':scheme'),
        u'wsgi.input': stream,
        u'wsgi.errors': sys.stderr,
        u'wsgi.multithread': True,
```

(continues on next page)

(continued from previous page)

```
        u'wsgi.multiprocess': False,
        u'wsgi.run_once': False,
    }
    if u'content-type' in header_dict:
        environ[u'CONTENT_TYPE'] = header_dict[u'content-type']
    if u'content-length' in header_dict:
        environ[u'CONTENT_LENGTH'] = header_dict[u'content-length']
    for name, value in header_dict.items():
        environ[u'HTTP_' + name.upper()] = value
    return environ


# Set up the WSGI app.
application_string = sys.argv[1]
path, func = application_string.split(':', 1)
module = importlib.import_module(path)
APPLICATION = getattr(module, func)

# Set up TLS
ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ssl_context.options |= (
    ssl.OP_NO_TLSv1 | ssl.OP_NO_TLSv1_1 | ssl.OP_NO_COMPRESSION
)
ssl_context.set_ciphers("ECDHE+AESGCM")
ssl_context.load_cert_chain(certfile="cert.crt", keyfile="cert.key")
ssl_context.set_alpn_protocols(["h2"])

# Do the asnycio bits
loop = asyncio.get_event_loop()
# Each client connection will create a new protocol instance
coro = loop.create_server(H2Protocol, '127.0.0.1', 8443, ssl=ssl_context)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

1.4.2 Example Clients

Plain Sockets Example Client

This example is a basic HTTP/2 client written using plain Python sockets, and ssl TLS/SSL wrapper for socket objects.

This client is *not* a complete production-ready HTTP/2 client and only intended as a demonstration sample.

This example shows the bare minimum that is needed to send an HTTP/2 request to a server, and read back a response body.

```python
#!/usr/bin/env python3

"""
plain_sockets_client.py
~~~~~~~~~~~~~~~~~~~~~~~

Just enough code to send a GET request via h2 to an HTTP/2 server and receive a response body.
This is *not* a complete production-ready HTTP/2 client!
"""

import socket
import ssl
import certifi

import h2.connection
import h2.events


SERVER_NAME = 'http2.golang.org'
SERVER_PORT = 443

# generic socket and ssl configuration
socket.setdefaulttimeout(15)
ctx = ssl.create_default_context(cafile=certifi.where())
ctx.set_alpn_protocols(['h2'])

# open a socket to the server and initiate TLS/SSL
s = socket.create_connection((SERVER_NAME, SERVER_PORT))
s = ctx.wrap_socket(s, server_hostname=SERVER_NAME)

c = h2.connection.H2Connection()
c.initiate_connection()
s.sendall(c.data_to_send())

headers = [
    (':method', 'GET'),
    (':path', '/reqinfo'),
    (':authority', SERVER_NAME),
    (':scheme', 'https'),
]
c.send_headers(1, headers, end_stream=True)
s.sendall(c.data_to_send())

body = b''
response_stream_ended = False
while not response_stream_ended:
    # read raw data from the socket
    data = s.recv(65536 * 1024)
    if not data:
        break

    # feed raw data into h2, and process resulting events
    events = c.receive_data(data)
    for event in events:
        print(event)
        if isinstance(event, h2.events.DataReceived):
```

(continues on next page)

(continued from previous page)

```
            # update flow control so the server doesn't starve us
            c.acknowledge_received_data(event.flow_controlled_length, event.stream_id)
            # more response body data received
            body += event.data
        if isinstance(event, h2.events.StreamEnded):
            # response body completed, let's exit the loop
            response_stream_ended = True
            break
    # send any pending data to the server
    s.sendall(c.data_to_send())

print("Response fully received:")
print(body.decode())

# tell the server we are closing the h2 connection
c.close_connection()
s.sendall(c.data_to_send())

# close the socket
s.close()
```

Twisted Example Client: Head Requests

This example is a basic HTTP/2 client written for the Twisted asynchronous networking framework.

This client is fairly simple: it makes a hard-coded HEAD request to nghttp2.org/httpbin/ and prints out the response data. Its purpose is to demonstrate how to write a very basic HTTP/2 client implementation.

```
# -*- coding: utf-8 -*-
"""
head_request.py
~~~~~~~~~~~~~~~

A short example that demonstrates a client that makes HEAD requests to certain
websites.

This example is intended as a reproduction of nghttp2 issue 396, for the
purposes of compatibility testing.
"""
from __future__ import print_function

from twisted.internet import reactor
from twisted.internet.endpoints import connectProtocol, SSL4ClientEndpoint
from twisted.internet.protocol import Protocol
from twisted.internet.ssl import optionsForClientTLS
from hyperframe.frame import SettingsFrame
from h2.connection import H2Connection
from h2.events import (
    ResponseReceived, DataReceived, StreamEnded,
    StreamReset, SettingsAcknowledged,
)


AUTHORITY = u'nghttp2.org'
PATH = '/httpbin/'
```

(continues on next page)

(continued from previous page)

```
SIZE = 4096


class H2Protocol(Protocol):
    def __init__(self):
        self.conn = H2Connection()
        self.known_proto = None
        self.request_made = False

    def connectionMade(self):
        self.conn.initiate_connection()

        # This reproduces the error in #396, by changing the header table size.
        self.conn.update_settings({SettingsFrame.HEADER_TABLE_SIZE: SIZE})

        self.transport.write(self.conn.data_to_send())

    def dataReceived(self, data):
        if not self.known_proto:
            self.known_proto = self.transport.negotiatedProtocol
            assert self.known_proto == b'h2'

        events = self.conn.receive_data(data)

        for event in events:
            if isinstance(event, ResponseReceived):
                self.handleResponse(event.headers, event.stream_id)
            elif isinstance(event, DataReceived):
                self.handleData(event.data, event.stream_id)
            elif isinstance(event, StreamEnded):
                self.endStream(event.stream_id)
            elif isinstance(event, SettingsAcknowledged):
                self.settingsAcked(event)
            elif isinstance(event, StreamReset):
                reactor.stop()
                raise RuntimeError("Stream reset: %d" % event.error_code)
            else:
                print(event)

        data = self.conn.data_to_send()
        if data:
            self.transport.write(data)

    def settingsAcked(self, event):
        # Having received the remote settings change, lets send our request.
        if not self.request_made:
            self.sendRequest()

    def handleResponse(self, response_headers, stream_id):
        for name, value in response_headers:
            print("%s: %s" % (name.decode('utf-8'), value.decode('utf-8')))

        print("")

    def handleData(self, data, stream_id):
        print(data, end='')

```

(continues on next page)

(continued from previous page)

```
    def endStream(self, stream_id):
        self.conn.close_connection()
        self.transport.write(self.conn.data_to_send())
        self.transport.loseConnection()
        reactor.stop()

    def sendRequest(self):
        request_headers = [
            (':method', 'HEAD'),
            (':authority', AUTHORITY),
            (':scheme', 'https'),
            (':path', PATH),
        ]
        self.conn.send_headers(1, request_headers, end_stream=True)
        self.request_made = True

options = optionsForClientTLS(
    hostname=AUTHORITY,
    acceptableProtocols=[b'h2'],
)

connectProtocol(
    SSL4ClientEndpoint(reactor, AUTHORITY, 443, options),
    H2Protocol()
)
reactor.run()
```

Twisted Example Client: Post Requests

This example is a basic HTTP/2 client written for the Twisted asynchronous networking framework.

This client is fairly simple: it makes a hard-coded POST request to nghttp2.org/httpbin/post and prints out the response data, sending a file that is provided on the command line or the script itself. Its purpose is to demonstrate how to write a HTTP/2 client implementation that handles flow control.

```
# -*- coding: utf-8 -*-
"""
post_request.py
~~~~~~~~~~~~~~~

A short example that demonstrates a client that makes POST requests to certain
websites.

This example is intended to demonstrate how to handle uploading request bodies.
In this instance, a file will be uploaded. In order to handle arbitrary files,
this example also demonstrates how to obey HTTP/2 flow control rules.

Takes one command-line argument: a path to a file in the filesystem to upload.
If none is present, uploads this file.
"""
from __future__ import print_function

import mimetypes
import os
import sys
```

(continues on next page)

(continued from previous page)

```

from twisted.internet import reactor, defer
from twisted.internet.endpoints import connectProtocol, SSL4ClientEndpoint
from twisted.internet.protocol import Protocol
from twisted.internet.ssl import optionsForClientTLS
from h2.connection import H2Connection
from h2.events import (
    ResponseReceived, DataReceived, StreamEnded, StreamReset, WindowUpdated,
    SettingsAcknowledged,
)


AUTHORITY = u'nghttp2.org'
PATH = '/httpbin/post'


class H2Protocol(Protocol):
    def __init__(self, file_path):
        self.conn = H2Connection()
        self.known_proto = None
        self.request_made = False
        self.request_complete = False
        self.file_path = file_path
        self.flow_control_deferred = None
        self.fileobj = None
        self.file_size = None

    def connectionMade(self):
        """
        Called by Twisted when the TCP connection is established. We can start
        sending some data now: we should open with the connection preamble.
        """
        self.conn.initiate_connection()
        self.transport.write(self.conn.data_to_send())

    def dataReceived(self, data):
        """
        Called by Twisted when data is received on the connection.

        We need to check a few things here. Firstly, we want to validate that
        we actually negotiated HTTP/2: if we didn't, we shouldn't proceed!

        Then, we want to pass the data to the protocol stack and check what
        events occurred.
        """
        if not self.known_proto:
            self.known_proto = self.transport.negotiatedProtocol
            assert self.known_proto == b'h2'

        events = self.conn.receive_data(data)

        for event in events:
            if isinstance(event, ResponseReceived):
                self.handleResponse(event.headers)
            elif isinstance(event, DataReceived):
                self.handleData(event.data)
            elif isinstance(event, StreamEnded):
```

(continues on next page)

(continued from previous page)

```
                self.endStream()
            elif isinstance(event, SettingsAcknowledged):
                self.settingsAcked(event)
            elif isinstance(event, StreamReset):
                reactor.stop()
                raise RuntimeError("Stream reset: %d" % event.error_code)
            elif isinstance(event, WindowUpdated):
                self.windowUpdated(event)

        data = self.conn.data_to_send()
        if data:
            self.transport.write(data)

    def settingsAcked(self, event):
        """
        Called when the remote party ACKs our settings. We send a SETTINGS
        frame as part of the preamble, so if we want to be very polite we can
        wait until the ACK for that frame comes before we start sending our
        request.
        """
        if not self.request_made:
            self.sendRequest()

    def handleResponse(self, response_headers):
        """
        Handle the response by printing the response headers.
        """
        for name, value in response_headers:
            print("%s: %s" % (name.decode('utf-8'), value.decode('utf-8')))

        print("")

    def handleData(self, data):
        """
        We handle data that's received by just printing it.
        """
        print(data, end='')

    def endStream(self):
        """
        We call this when the stream is cleanly ended by the remote peer. That
        means that the response is complete.

        Because this code only makes a single HTTP/2 request, once we receive
        the complete response we can safely tear the connection down and stop
        the reactor. We do that as cleanly as possible.
        """
        self.request_complete = True
        self.conn.close_connection()
        self.transport.write(self.conn.data_to_send())
        self.transport.loseConnection()

    def windowUpdated(self, event):
        """
        We call this when the flow control window for the connection or the
        stream has been widened. If there's a flow control deferred present
        (that is, if we're blocked behind the flow control), we fire it.
```

(continues on next page)

(continued from previous page)

```
        Otherwise, we do nothing.
        """
        if self.flow_control_deferred is None:
            return

        # Make sure we remove the flow control deferred to avoid firing it
        # more than once.
        flow_control_deferred = self.flow_control_deferred
        self.flow_control_deferred = None
        flow_control_deferred.callback(None)

    def connectionLost(self, reason=None):
        """
        Called by Twisted when the connection is gone. Regardless of whether
        it was clean or not, we want to stop the reactor.
        """
        if self.fileobj is not None:
            self.fileobj.close()

        if reactor.running:
            reactor.stop()

    def sendRequest(self):
        """
        Send the POST request.

        A POST request is made up of one headers frame, and then 0+ data
        frames. This method begins by sending the headers, and then starts a
        series of calls to send data.
        """
        # First, we need to work out how large the file is.
        self.file_size = os.stat(self.file_path).st_size

        # Next, we want to guess a content-type and content-encoding.
        content_type, content_encoding = mimetypes.guess_type(self.file_path)

        # Now we can build a header block.
        request_headers = [
            (':method', 'POST'),
            (':authority', AUTHORITY),
            (':scheme', 'https'),
            (':path', PATH),
            ('content-length', str(self.file_size)),
        ]

        if content_type is not None:
            request_headers.append(('content-type', content_type))

            if content_encoding is not None:
                request_headers.append(('content-encoding', content_encoding))

        self.conn.send_headers(1, request_headers)
        self.request_made = True

        # We can now open the file.
        self.fileobj = open(self.file_path, 'rb')

```

(continues on next page)

(continued from previous page)

```
        # We now need to send all the relevant data. We do this by checking
        # what the acceptable amount of data is to send, and sending it. If we
        # find ourselves blocked behind flow control, we then place a deferred
        # and wait until that deferred fires.
        self.sendFileData()

    def sendFileData(self):
        """
        Send some file data on the connection.
        """
        # Firstly, check what the flow control window is for stream 1.
        window_size = self.conn.local_flow_control_window(stream_id=1)

        # Next, check what the maximum frame size is.
        max_frame_size = self.conn.max_outbound_frame_size

        # We will send no more than the window size or the remaining file size
        # of data in this call, whichever is smaller.
        bytes_to_send = min(window_size, self.file_size)

        # We now need to send a number of data frames.
        while bytes_to_send > 0:
            chunk_size = min(bytes_to_send, max_frame_size)
            data_chunk = self.fileobj.read(chunk_size)
            self.conn.send_data(stream_id=1, data=data_chunk)

            bytes_to_send -= chunk_size
            self.file_size -= chunk_size

        # We've prepared a whole chunk of data to send. If the file is fully
        # sent, we also want to end the stream: we're done here.
        if self.file_size == 0:
            self.conn.end_stream(stream_id=1)
        else:
            # We've still got data left to send but the window is closed. Save
            # a Deferred that will call us when the window gets opened.
            self.flow_control_deferred = defer.Deferred()
            self.flow_control_deferred.addCallback(self.sendFileData)

        self.transport.write(self.conn.data_to_send())


try:
    filename = sys.argv[1]
except IndexError:
    filename = __file__

options = optionsForClientTLS(
    hostname=AUTHORITY,
    acceptableProtocols=[b'h2'],
)

connectProtocol(
    SSL4ClientEndpoint(reactor, AUTHORITY, 443, options),
    H2Protocol(filename)
)
reactor.run()
```

1.5 Advanced Usage

1.5.1 Priority

New in version 2.0.0.

RFC 7540 has a fairly substantial and complex section describing how to build a HTTP/2 priority tree, and the effect that should have on sending data from a server.

h2 does not enforce any priority logic by default for servers. This is because scheduling data sends is outside the scope of this library, as it likely requires fairly substantial understanding of the scheduler being used.

However, for servers that *do* want to follow the priority recommendations given by clients, the Hyper project provides an implementation of the RFC 7540 priority tree that will be useful to plug into a server. That, combined with the `PriorityUpdated` event from this library, can be used to build a server that conforms to RFC 7540's recommendations for priority handling.

1.5.2 Related Events

New in version 2.4.0.

In the 2.4.0 release h2 added support for signaling "related events". These are a HTTP/2-only construct that exist because certain HTTP/2 events can occur simultaneously: that is, one HTTP/2 frame can cause multiple state transitions to occur at the same time. One example of this is a HEADERS frame that contains priority information and carries the END_STREAM flag: this would cause three events to fire (one of the various request/response received events, a `PriorityUpdated` event, and a `StreamEnded` event).

Ordinarily h2's logic will emit those events to you one at a time. This means that you may attempt to process, for example, a `DataReceived` event, not knowing that the next event out will be a `StreamEnded` event. h2 *does* know this, however, and so will forbid you from taking certain actions that are a violation of the HTTP/2 protocol.

To avoid this asymmetry of information, events that can occur simultaneously now carry properties for their "related events". These allow users to find the events that can have occurred simultaneously with each other before the event is emitted by h2. The following objects have "related events":

- `RequestReceived`:
  - `stream_ended`: any `StreamEnded` event that occurred at the same time as receiving this request.
  - `priority_updated`: any `PriorityUpdated` event that occurred at the same time as receiving this request.
- `ResponseReceived`:
  - `stream_ended`: any `StreamEnded` event that occurred at the same time as receiving this response.
  - `priority_updated`: any `PriorityUpdated` event that occurred at the same time as receiving this response.
- `TrailersReceived`:
  - `stream_ended`: any `StreamEnded` event that occurred at the same time as receiving this set of trailers. This will **always** be present for trailers, as they must terminate streams.
  - `priority_updated`: any `PriorityUpdated` event that occurred at the same time as receiving this response.
- `InformationalResponseReceived`:
  - `priority_updated`: any `PriorityUpdated` event that occurred at the same time as receiving this informational response.

- `DataReceived`:
  - `stream_ended`: any `StreamEnded` event that occurred at the same time as receiving this data.

> **Warning:** h2 does not know if you are looking for related events or expecting to find events in the event stream. Therefore, it will always emit "related events" in the event stream. If you are using the "related events" event pattern, you will want to be careful to avoid double-processing related events.

1.5.3 Connections: Advanced

Thread Safety

`H2Connection` objects are *not* thread-safe. They cannot safely be accessed from multiple threads at once. This is a deliberate design decision: it is not trivially possible to design the `H2Connection` object in a way that would be either lock-free or have the locks at a fine granularity.

Your implementations should bear this in mind, and handle it appropriately. It should be simple enough to use locking alongside the `H2Connection`: simply lock around the connection object itself. Because the `H2Connection` object does no I/O it should be entirely safe to do that. Alternatively, have a single thread take ownership of the `H2Connection` and use a message-passing interface to serialize access to the `H2Connection`.

If you are using a non-threaded concurrency approach (e.g. Twisted), this should not affect you.

Internal Buffers

In order to avoid doing I/O, the `H2Connection` employs an internal buffer. This buffer is *unbounded* in size: it can potentially grow infinitely. This means that, if you are not making sure to regularly empty it, you are at risk of exceeding the memory limit of a single process and finding your program crashes.

It is highly recommended that you send data at regular intervals, ideally as soon as possible.

Sending Data

When sending data on the network, it's important to remember that you may not be able to send an unbounded amount of data at once. Particularly when using TCP, it is often the case that there are limits on how much data may be in flight at any one time. These limits can be very low, and your operating system will only buffer so much data in memory before it starts to complain.

For this reason, it is possible to consume only a subset of the data available when you call `data_to_send`. However, once you have pulled the data out of the `H2Connection` internal buffer, it is *not* possible to put it back on again. For that reason, it is adviseable that you confirm how much space is available in the OS buffer before sending.

Alternatively, use tools made available by your framework. For example, the Python standard library `socket` module provides a `sendall` method that will automatically block until all the data has been sent. This will enable you to always use the unbounded form of `data_to_send`, and will help you avoid subtle bugs.

When To Send

In addition to knowing how much data to send (see *Sending Data*) it is important to know when to send data. For h2, this amounts to knowing when to call `data_to_send`.

h2 may write data into its send buffer at two times. The first is whenever `receive_data` is called. This data is sent in response to some control frames that require no user input: for example, responding to PING frames. The second time is in response to user action: whenever a user calls a method like *send_headers*, data may be written into the buffer.

In a standard design for a h2 consumer, then, that means there are two places where you'll potentially want to send data. The first is in your "receive data" loop. This is where you take the data you receive, pass it into *receive_data*, and then dispatch events. For this loop, it is usually best to save sending data until the loop is complete: that allows you to empty the buffer only once.

The other place you'll want to send the data is when initiating requests or taking any other active, unprompted action on the connection. In this instance, you'll want to make all the relevant `send_*` calls, and *then* call *data_to_send*.

1.5.4 Headers

HTTP/2 defines several "special header fields" which are used to encode data that was previously sent in either the request or status line of HTTP/1.1. These header fields are distinguished from ordinary header fields because their field name begins with a `:` character. The special header fields defined in RFC 7540 are:

- `:status`
- `:path`
- `:method`
- `:scheme`
- `:authority`

RFC 7540 **mandates** that all of these header fields appear *first* in the header block, before the ordinary header fields. This could cause difficulty if the *send_headers* method accepted a plain `dict` for the `headers` argument, because `dict` objects are unordered. For this reason, we require that you provide a list of two-tuples.

1.5.5 Flow Control

HTTP/2 defines a complex flow control system that uses a sliding window of data on both a per-stream and per-connection basis. Essentially, each implementation allows its peer to send a specific amount of data at any time (the "flow control window") before it must stop. Each stream has a separate window, and the connection as a whole has a window. Each window can be opened by an implementation by sending a `WINDOW_UPDATE` frame, either on a specific stream (causing the window for that stream to be opened), or on stream `0`, which causes the window for the entire connection to be opened.

In HTTP/2, only data in `DATA` frames is flow controlled. All other frames are exempt from flow control. Each `DATA` frame consumes both stream and connection flow control window bytes. This means that the maximum amount of data that can be sent on any one stream before a `WINDOW_UPDATE` frame is received is the *lower* of the stream and connection windows. The maximum amount of data that can be sent on *all* streams before a `WINDOW_UPDATE` frame is received is the size of the connection flow control window.

Working With Flow Control

The amount of flow control window a `DATA` frame consumes is the sum of both its contained application data *and* the amount of padding used. h2 shows this to the user in a *DataReceived* event by using the *flow_controlled_length* field. When working with flow control in h2, users *must* use this field: simply using `len(datareceived.data)` can eventually lead to deadlock.

When data has been received and given to the user in a *DataReceived*, it is the responsibility of the user to re-open the flow control window when the user is ready for more data. h2 does not do this automatically to avoid flooding the user with data: if we did, the remote peer could send unbounded amounts of data that the user would need to buffer before processing.

To re-open the flow control window, then, the user must call `increment_flow_control_window` with the `flow_controlled_length` of the received data. h2 requires that you manage both the connection and the stream flow control windows separately, so you may need to increment both the stream the data was received on and stream `0`.

When sending data, a HTTP/2 implementation must not send more than flow control window available for that stream. As noted above, the maximum amount of data that can be sent on the stream is the minimum of the stream and the connection flow control windows. You can find out how much data you can send on a given stream by using the `local_flow_control_window` method, which will do all of these calculations for you. If you attempt to send more than this amount of data on a stream, h2 will throw a `ProtocolError` and refuse to send the data.

In h2, receiving a `WINDOW_UPDATE` frame causes a `WindowUpdated` event to fire. This will notify you that there is potentially more room in a flow control window. Note that, just because an increment of a given size was received *does not* mean that that much more data can be sent: remember that both the connection and stream flow control windows constrain how much data can be sent.

As a result, when a `WindowUpdated` event fires with a non-zero stream ID, and the user has more data to send on that stream, the user should call `local_flow_control_window` to check if there really is more room to send data on that stream.

When a `WindowUpdated` event fires with a stream ID of `0`, that may have unblocked *all* streams that are currently blocked. The user should use `local_flow_control_window` to check all blocked streams to see if more data is available.

Auto Flow Control

New in version 2.5.0.

In most cases, there is no advantage for users in managing their own flow control strategies. While particular high performance or specific-use-case applications may gain value from directly controlling the emission of `WINDOW_UPDATE` frames, the average application can use a lowest-common-denominator strategy to emit those frames. As of version 2.5.0, h2 now provides this automatic strategy for users, if they want to use it.

This automatic strategy is built around a single method: `acknowledge_received_data`. This method flags to the connection object that your application has dealt with a certain number of flow controlled bytes, and that the window should be incremented in some way. Whenever your application has “processed” some received bytes, this method should be called to signal that they have been processed.

The key difference between this method and `increment_flow_control_window` is that the method `acknowledge_received_data` does not guarantee that it will emit a `WINDOW_UPDATE` frame, and if it does it will not necessarily emit them for *only* the stream or *only* the frame. Instead, the `WINDOW_UPDATE` frames will be *coalesced*: they will be emitted only when a certain number of bytes have been freed up.

For most applications, this method should be preferred to the manual flow control mechanism.

1.6 Low-Level Details

> **Warning:** This section of the documentation covers low-level implementation details of h2. This is most likely to be of use to h2 developers and to other HTTP/2 implementers, though it could well be of general interest. Feel free to peruse it, but if you’re looking for information about how to *use* h2 you should consider looking elsewhere.

1.6.1 State Machines

h2 is fundamentally built on top of a pair of interacting Finite State Machines. One of these FSMs manages per-connection state, and another manages per-stream state. Almost without exception (see *Priority* for more details) every single frame is unconditionally translated into events for both state machines and those state machines are turned.

The advantages of a system such as this is that the finite state machines can very densely encode the kinds of things that are allowed at any particular moment in a HTTP/2 connection. However, most importantly, almost all protocols are defined *in terms* of finite state machines: that is, protocol descriptions can be reduced to a number of states and inputs. That makes FSMs a very natural tool for implementing protocol stacks.

Indeed, most protocol implementations that do not explicitly encode a finite state machine almost always *implicitly* encode a finite state machine, by using classes with a bunch of variables that amount to state-tracking variables, or by using the call-stack as an implicit state tracking mechanism. While these methods are not immediately problematic, they tend to lack *explicitness*, and can lead to subtle bugs of the form "protocol action X is incorrectly allowed in state Y".

For these reasons, we have implemented two *explicit* finite state machines. These machines aim to encode most of the protocol-specific state, in particular regarding what frame is allowed at what time. This target goal is sometimes not achieved: in particular, as of this writing the *stream* FSM contains a number of other state variables that really ought to be rolled into the state machine itself in the form of new states, or in the form of a transformation of the FSM to use state *vectors* instead of state *scalars*.

The following sections contain some implementers notes on these FSMs.

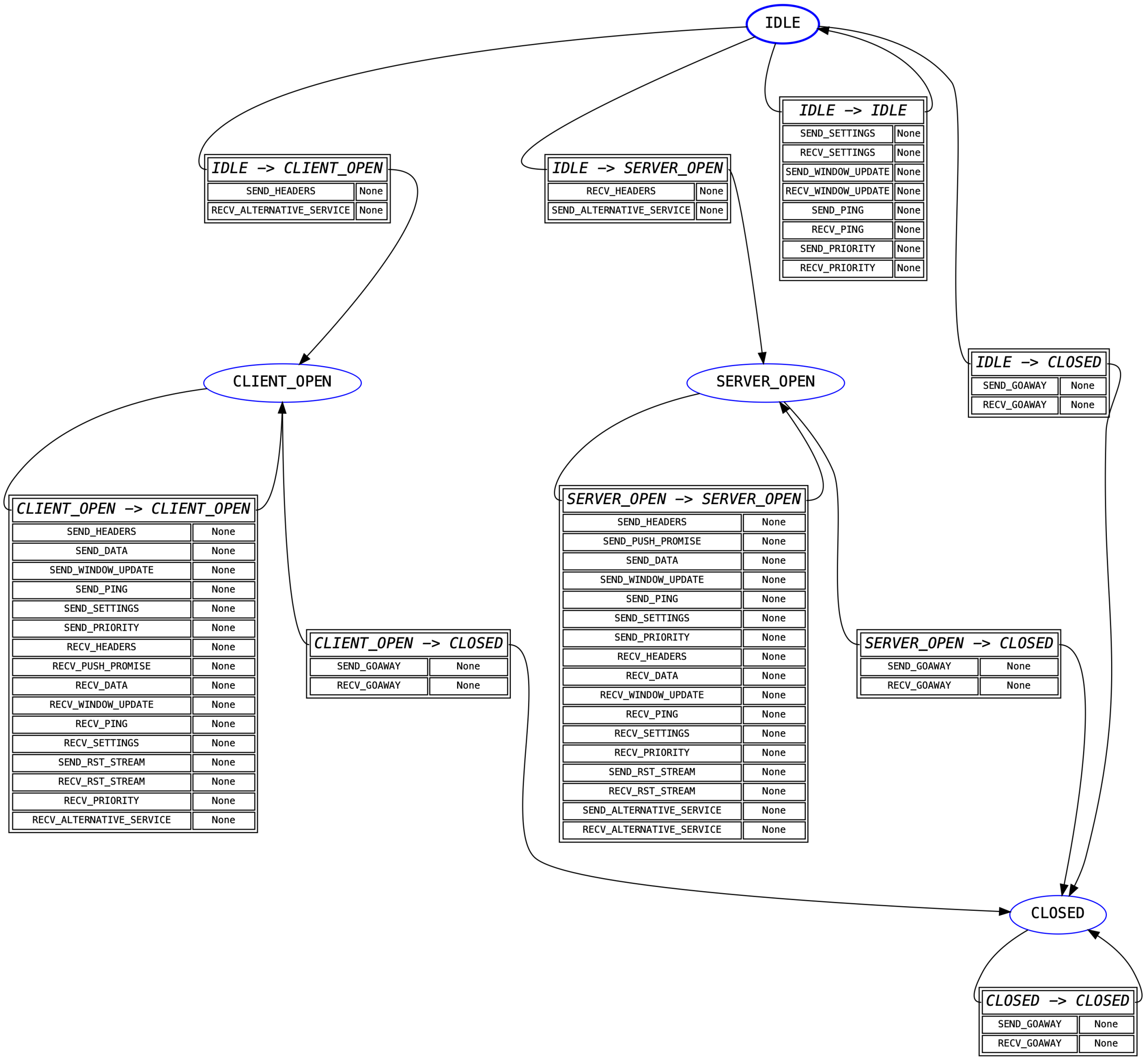
Connection State Machine

The "outer" state machine, the first one that is encountered when sending or receiving data, is the connection state machine. This state machine tracks whole-connection state.

This state machine is primarily intended to forbid certain actions on the basis of whether the implementation is acting as a client or a server. For example, clients are not permitted to send `PUSH_PROMISE` frames: this state machine forbids that by refusing to define a valid transition from the `CLIENT_OPEN` state for the `SEND_PUSH_PROMISE` event.

Otherwise, this particular state machine triggers no side-effects. It has a very coarse, high-level, functionality.

A visual representation of this FSM is shown below:



Stream State Machine

Once the connection state machine has been spun, any frame that belongs to a stream is passed to the stream state machine for its given stream. Each stream has its own instance of the state machine, but all of them share the transition table: this is because the table itself is sufficiently large that having it be per-instance would be a ridiculous memory overhead.

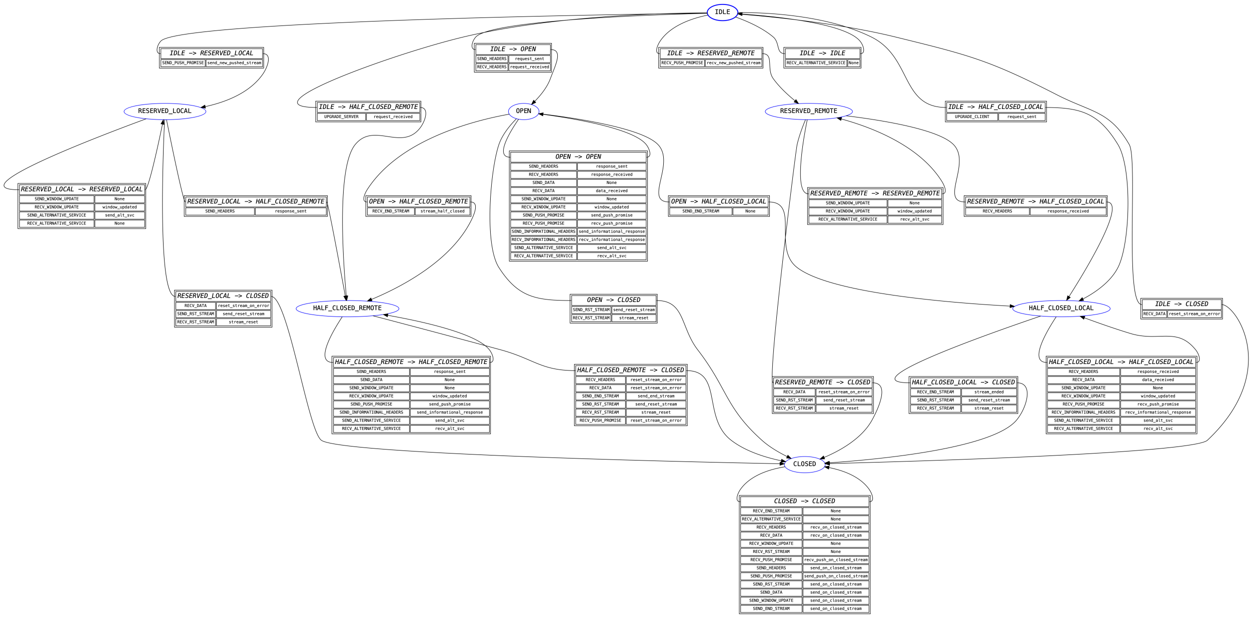
Unlike the connection state machine, the stream state machine is quite complex. This is because it frequently needs to encode some side-effects. The most common side-effect is emitting a `RST_STREAM` frame when an error is encountered: the need to do this means that far more transitions need to be encoded than for the connection state machine.

Many of the side-effect functions in this state machine also raise `ProtocolError` exceptions. This is almost always done on the basis of an extra state variable, which is an annoying code smell: it should always be possible for the state machine itself to police these using explicit state management. A future refactor will hopefully address this problem by making these additional state variables part of the state definitions in the FSM, which will lead to an expansion of the number of states but a greater degree of simplicity in understanding and tracking what is going on in the state machine.

The other action taken by the side-effect functions defined here is returning *events*. Most of these events are returned directly to the user, and reflect the specific state transition that has taken place, but some of the events are purely *internal*: they are used to signal to other parts of the h2 codebase what action has been taken.

The major use of the internal events functionality at this time is for validating header blocks: there are different rules for request headers than there are for response headers, and different rules again for trailers. The internal events are used to determine *exactly what* kind of data the user is attempting to send, and using that information to do the correct kind of validation. This approach ensures that the final source of truth about what's happening at the protocol level lives inside the FSM, which is an extremely important design principle we want to continue to enshrine in h2.

A visual representation of this FSM is shown below:


IDLE
IDLE -> RESERVED_LOCAL
SEND_PUSH_PROMISE send_new_pushed_stream
IDLE -> OPEN
SEND_HEADERS request_sent
RECV_HEADERS request_received
IDLE -> RESERVED_REMOTE
RECV_PUSH_PROMISE recv_new_pushed_stream
IDLE -> IDLE
RECV_ALTERNATIVE_SERVICE None
RESERVED_LOCAL
IDLE -> HALF_CLOSED_REMOTE
UPGRADE_SERVER request_received
OPEN
RESERVED_REMOTE
IDLE -> HALF_CLOSED_LOCAL
UPGRADE_CLIENT request_sent
OPEN -> OPEN
SEND_HEADERS response_sent
RECV_HEADERS response_received
SEND_DATA None
RECV_DATA data_received
SEND_WINDOW_UPDATE None
RECV_WINDOW_UPDATE window_updated
SEND_PUSH_PROMISE send_push_promise
RECV_PUSH_PROMISE recv_push_promise
SEND_INFORMATIONAL_HEADERS send_informational_response
RECV_INFORMATIONAL_HEADERS recv_informational_response
SEND_ALTERNATIVE_SERVICE send_alt_svc
RECV_ALTERNATIVE_SERVICE recv_alt_svc
RESERVED_LOCAL -> RESERVED_LOCAL
SEND_WINDOW_UPDATE None
RECV_WINDOW_UPDATE window_updated
SEND_ALTERNATIVE_SERVICE send_alt_svc
RECV_ALTERNATIVE_SERVICE None
RESERVED_LOCAL -> HALF_CLOSED_REMOTE
SEND_HEADERS response_sent
OPEN -> HALF_CLOSED_REMOTE
RECV_END_STREAM stream_half_closed
OPEN -> HALF_CLOSED_LOCAL
SEND_END_STREAM None
RESERVED_REMOTE -> RESERVED_REMOTE
SEND_WINDOW_UPDATE None
RECV_WINDOW_UPDATE window_updated
RECV_ALTERNATIVE_SERVICE recv_alt_svc
RESERVED_REMOTE -> HALF_CLOSED_LOCAL
RECV_HEADERS response_received
RESERVED_LOCAL -> CLOSED
RECV_DATA reset_stream_on_error
SEND_RST_STREAM send_reset_stream
RECV_RST_STREAM stream_reset
HALF_CLOSED_REMOTE
OPEN -> CLOSED
SEND_RST_STREAM send_reset_stream
RECV_RST_STREAM stream_reset
HALF_CLOSED_LOCAL
IDLE -> CLOSED
RECV_DATA reset_stream_on_error
HALF_CLOSED_REMOTE -> HALF_CLOSED_REMOTE
SEND_HEADERS response_sent
SEND_DATA None
SEND_WINDOW_UPDATE None
RECV_WINDOW_UPDATE window_updated
SEND_PUSH_PROMISE send_push_promise
SEND_INFORMATIONAL_HEADERS send_informational_response
SEND_ALTERNATIVE_SERVICE send_alt_svc
RECV_ALTERNATIVE_SERVICE recv_alt_svc
HALF_CLOSED_REMOTE -> CLOSED
RECV_HEADERS reset_stream_on_error
RECV_DATA reset_stream_on_error
SEND_END_STREAM send_end_stream
SEND_RST_STREAM send_reset_stream
RECV_RST_STREAM stream_reset
RECV_PUSH_PROMISE reset_stream_on_error
RESERVED_REMOTE -> CLOSED
RECV_DATA reset_stream_on_error
SEND_RST_STREAM send_reset_stream
RECV_RST_STREAM stream_reset
HALF_CLOSED_LOCAL -> CLOSED
RECV_END_STREAM stream_ended
SEND_RST_STREAM send_reset_stream
RECV_RST_STREAM stream_reset
HALF_CLOSED_LOCAL -> HALF_CLOSED_LOCAL
RECV_HEADERS response_received
RECV_DATA data_received
SEND_WINDOW_UPDATE None
RECV_WINDOW_UPDATE window_updated
RECV_PUSH_PROMISE recv_push_promise
RECV_INFORMATIONAL_HEADERS recv_informational_response
SEND_ALTERNATIVE_SERVICE send_alt_svc
RECV_ALTERNATIVE_SERVICE recv_alt_svc
CLOSED
CLOSED -> CLOSED
RECV_END_STREAM None
RECV_ALTERNATIVE_SERVICE None
RECV_HEADERS recv_on_closed_stream
RECV_DATA recv_on_closed_stream
RECV_WINDOW_UPDATE None
RECV_RST_STREAM None
RECV_PUSH_PROMISE recv_push_on_closed_stream
SEND_HEADERS send_on_closed_stream
SEND_PUSH_PROMISE send_push_on_closed_stream
SEND_RST_STREAM send_on_closed_stream
SEND_DATA send_on_closed_stream
SEND_WINDOW_UPDATE send_on_closed_stream
SEND_END_STREAM send_on_closed_stream

Priority

In the *Stream State Machine* section we said that any frame that belongs to a stream is passed to the stream state machine. This turns out to be not quite true.

Specifically, while `PRIORITY` frames are technically sent on a given stream (that is, RFC 7540 Section 6.3 defines them as "always identifying a stream" and forbids the use of stream ID `0` for them), in practice they are almost completely exempt from the usual stream FSM behaviour. Specifically, the RFC has this to say:

> The `PRIORITY` frame can be sent on a stream in any state, though it cannot be sent between consecutive frames that comprise a single header block (Section 4.3).

Given that the consecutive header block requirement is handled outside of the FSMs, this section of the RFC essentially means that there is *never* a situation where it is invalid to receive a `PRIORITY` frame. This means that including it in the stream FSM would require that we allow `SEND_PRIORITY` and `RECV_PRIORITY` in all states.

This is not a totally onerous task: however, another key note is that h2 uses the *absence* of a stream state machine to flag a closed stream. This is primarily for memory conservation reasons: if we needed to keep around an FSM for every stream we've ever seen, that would cause long-lived HTTP/2 connections to consume increasingly large amounts of memory. On top of this, it would require us to create a stream FSM each time we received a `PRIORITY` frame for a given stream, giving a malicious peer an easy route to force a h2 user to allocate nearly unbounded amounts of memory.

For this reason, h2 circumvents the stream FSM entirely for PRIORITY frames. Instead, these frames are treated as being connection-level frames that *just happen* to identify a specific stream. They do not bring streams into being, or in any sense interact with h2's view of streams. Their stream details are treated as strictly metadata that h2 is not interested in beyond being able to parse it out.

1.7 h2 API

This document details the API of h2.

1.7.1 Semantic Versioning

h2 follows semantic versioning for its public API. Please note that the guarantees of semantic versioning apply only to the API that is *documented here*. Simply because a method or data field is not prefaced by an underscore does not make it part of h2's public API. Anything not documented here is subject to change at any time.

1.7.2 Connection

class h2.connection.**H2Connection**(*config=None*)

A low-level HTTP/2 connection object. This handles building and receiving frames and maintains both connection and per-stream state for all streams on this connection.

This wraps a HTTP/2 Connection state machine implementation, ensuring that frames can only be sent/received when the connection is in a valid state. It also builds stream state machines on demand to ensure that the constraints of those state machines are met as well. Attempts to create frames that cannot be sent will raise a ProtocolError.

Changed in version 2.3.0: Added the header_encoding keyword argument.

Changed in version 2.5.0: Added the config keyword argument. Deprecated the client_side and header_encoding parameters.

Changed in version 3.0.0: Removed deprecated parameters and properties.

Parameters **config** (*H2Configuration*) – The configuration for the HTTP/2 connection.

New in version 2.5.0.

acknowledge_received_data(*acknowledged_size, stream_id*)

Inform the *H2Connection* that a certain number of flow-controlled bytes have been processed, and that the space should be handed back to the remote peer at an opportune time.

New in version 2.5.0.

Parameters

- **acknowledged_size** (int) – The total *flow-controlled size* of the data that has been processed. Note that this must include the amount of padding that was sent with that data.
- **stream_id** (int) – The ID of the stream on which this data was received.

Returns Nothing

Return type None

advertise_alternative_service(*field_value, origin=None, stream_id=None*)

Notify a client about an available Alternative Service.

An Alternative Service is defined in RFC 7838. An Alternative Service notification informs a client that a given origin is also available elsewhere.

Alternative Services can be advertised in two ways. Firstly, they can be advertised explicitly: that is, a server can say "origin X is also available at Y". To advertise like this, set the `origin` argument and not the `stream_id` argument. Alternatively, they can be advertised implicitly: that is, a server can say "the origin you're contacting on stream X is also available at Y". To advertise like this, set the `stream_id` argument and not the `origin` argument.

The explicit method of advertising can be done as long as the connection is active. The implicit method can only be done after the client has sent the request headers and before the server has sent the response headers: outside of those points, h2 will forbid sending the Alternative Service advertisement by raising a ProtocolError.

The `field_value` parameter is specified in RFC 7838. h2 does not validate or introspect this argument: the user is required to ensure that it's well-formed. `field_value` corresponds to RFC 7838's "Alternative Service Field Value".

Note: It is strongly preferred to use the explicit method of advertising Alternative Services. The implicit method of advertising Alternative Services has a number of subtleties and can lead to inconsistencies between the server and client. h2 allows both mechanisms, but caution is strongly advised.

New in version 2.3.0.

Parameters

- **`field_value`** (`bytes`) – The RFC 7838 Alternative Service Field Value. This argument is not introspected by h2: the user is responsible for ensuring that it is well-formed.
- **`origin`** (`bytes` or `None`) – The origin/authority to which the Alternative Service being advertised applies. Must not be provided at the same time as `stream_id`.
- **`stream_id`** (`int` or `None`) – The ID of the stream which was sent to the authority for which this Alternative Service advertisement applies. Must not be provided at the same time as `origin`.

Returns Nothing.

`clear_outbound_data_buffer``()`

Clears the outbound data buffer, such that if this call was immediately followed by a call to *`data_to_send`*, that call would return no data.

This method should not normally be used, but is made available to avoid exposing implementation details.

`close_connection`(*error_code=0*, *additional_data=None*, *last_stream_id=None*)

Close a connection, emitting a GOAWAY frame.

Changed in version 2.4.0: Added `additional_data` and `last_stream_id` arguments.

Parameters

- **`error_code`** – (optional) The error code to send in the GOAWAY frame.
- **`additional_data`** – (optional) Additional debug data indicating a reason for closing the connection. Must be a bytestring.
- **`last_stream_id`** – (optional) The last stream which was processed by the sender. Defaults to `highest_inbound_stream_id`.

Returns Nothing

config = None

The configuration for this HTTP/2 connection object.

New in version 2.5.0.

data_to_send(*amount=None*)

Returns some data for sending out of the internal data buffer.

This method is analogous to `read` on a file-like object, but it doesn't block. Instead, it returns as much data as the user asks for, or less if that much data is not available. It does not perform any I/O, and so uses a different name.

Parameters **amount** (`int`) – (optional) The maximum amount of data to return. If not set, or set to `None`, will return as much data as possible.

Returns A bytestring containing the data to send on the wire.

Return type `bytes`

end_stream(*stream_id*)

Cleanly end a given stream.

This method ends a stream by sending an empty DATA frame on that stream with the `END_STREAM` flag set.

Parameters **stream_id** (`int`) – The ID of the stream to end.

Returns Nothing

get_next_available_stream_id()

Returns an integer suitable for use as the stream ID for the next stream created by this endpoint. For server endpoints, this stream ID will be even. For client endpoints, this stream ID will be odd. If no stream IDs are available, raises *NoAvailableStreamIDError*.

> **Warning:** The return value from this function does not change until the stream ID has actually been used by sending or pushing headers on that stream. For that reason, it should be called as close as possible to the actual use of the stream ID.

New in version 2.0.0.

Raises *NoAvailableStreamIDError*

Returns The next free stream ID this peer can use to initiate a stream.

Return type `int`

increment_flow_control_window(*increment*, *stream_id=None*)

Increment a flow control window, optionally for a single stream. Allows the remote peer to send more data.

Changed in version 2.0.0: Rejects attempts to increment the flow control window by out of range values with a `ValueError`.

Parameters

- **increment** (`int`) – The amount to increment the flow control window by.
- **stream_id** (`int` or `None`) – (optional) The ID of the stream that should have its flow control window opened. If not present or `None`, the connection flow control window will be opened instead.

Returns Nothing

Raises `ValueError`

`initiate_connection``()`

Provides any data that needs to be sent at the start of the connection. Must be called for both clients and servers.

`initiate_upgrade_connection``(`*settings_header=None*`)`

Call to initialise the connection object for use with an upgraded HTTP/2 connection (i.e. a connection negotiated using the `Upgrade: h2c` HTTP header).

This method differs from *`initiate_connection`* in several ways. Firstly, it handles the additional SETTINGS frame that is sent in the `HTTP2-Settings` header field. When called on a client connection, this method will return a bytestring that the caller can put in the `HTTP2-Settings` field they send on their initial request. When called on a server connection, the user **must** provide the value they received from the client in the `HTTP2-Settings` header field to the `settings_header` argument, which will be used appropriately.

Additionally, this method sets up stream 1 in a half-closed state appropriate for this side of the connection, to reflect the fact that the request is already complete.

Finally, this method also prepares the appropriate preamble to be sent after the upgrade.

New in version 2.3.0.

Parameters **`settings_header`** (`bytes`) – (optional, server-only): The value of the `HTTP2-Settings` header field received from the client.

Returns For clients, a bytestring to put in the `HTTP2-Settings`. For servers, returns nothing.

Return type `bytes` or `None`

`local_flow_control_window``(`*stream_id*`)`

Returns the maximum amount of data that can be sent on stream `stream_id`.

This value will never be larger than the total data that can be sent on the connection: even if the given stream allows more data, the connection window provides a logical maximum to the amount of data that can be sent.

The maximum data that can be sent in a single data frame on a stream is either this value, or the maximum frame size, whichever is *smaller*.

Parameters **`stream_id`** (`int`) – The ID of the stream whose flow control window is being queried.

Returns The amount of data in bytes that can be sent on the stream before the flow control window is exhausted.

Return type `int`

`max_inbound_frame_size = None`

The maximum size of a frame that can be received by this peer, in bytes.

`max_outbound_frame_size = None`

The maximum size of a frame that can be emitted by this peer, in bytes.

`open_inbound_streams`

The current number of open inbound streams.

`open_outbound_streams`

The current number of open outbound streams.

`ping``(`*opaque_data*`)`

Send a PING frame.

Parameters **opaque_data** – A bytestring of length 8 that will be sent in the PING frame.

Returns Nothing

prioritize(*stream_id*, *weight=None*, *depends_on=None*, *exclusive=None*)

Notify a server about the priority of a stream.

Stream priorities are a form of guidance to a remote server: they inform the server about how important a given response is, so that the server may allocate its resources (e.g. bandwidth, CPU time, etc.) accordingly. This exists to allow clients to ensure that the most important data arrives earlier, while less important data does not starve out the more important data.

Stream priorities are explained in depth in RFC 7540 Section 5.3.

This method updates the priority information of a single stream. It may be called well before a stream is actively in use, or well after a stream is closed.

> **Warning:** RFC 7540 allows for servers to change the priority of streams. However, h2 **does not** allow server stacks to do this. This is because most clients do not adequately know how to respond when provided conflicting priority information, and relatively little utility is provided by making that functionality available.

> **Note:** h2 **does not** maintain any information about the RFC 7540 priority tree. That means that h2 does not prevent incautious users from creating invalid priority trees, particularly by creating priority loops. While some basic error checking is provided by h2, users are strongly recommended to understand their prioritisation strategies before using the priority tools here.

> **Note:** Priority information is strictly advisory. Servers are allowed to disregard it entirely. Avoid relying on the idea that your priority signaling will definitely be obeyed.

New in version 2.4.0.

Parameters

- **stream_id** (`int`) – The ID of the stream to prioritize.
- **weight** (`int`) – The weight to give the stream. Defaults to `16`, the default weight of any stream. May be any value between `1` and `256` inclusive. The relative weight of a stream indicates what proportion of available resources will be allocated to that stream.
- **depends_on** (`int`) – The ID of the stream on which this stream depends. This stream will only be progressed if it is impossible to progress the parent stream (the one on which this one depends). Passing the value `0` means that this stream does not depend on any other. Defaults to `0`.
- **exclusive** (`bool`) – Whether this stream is an exclusive dependency of its “parent” stream (i.e. the stream given by `depends_on`). If a stream is an exclusive dependency of another, that means that all previously-set children of the parent are moved to become children of the new exclusively-dependent stream. Defaults to `False`.

push_stream(*stream_id*, *promised_stream_id*, *request_headers*)

Push a response to the client by sending a PUSH_PROMISE frame.

If it is important to send HPACK “never indexed” header fields (as defined in RFC 7451 Section 7.1.3), the user may instead provide headers using the HPACK library’s `HeaderTuple` and `NeverIndexedHeaderTuple` objects.

Parameters

- **stream_id** (int) – The ID of the stream that this push is a response to.
- **promised_stream_id** (int) – The ID of the stream that the pushed response will be sent on.
- **request_headers** (An iterable of two tuples of bytestrings or HeaderTuple objects.) – The headers of the request that the pushed response will be responding to.

Returns Nothing

receive_data(*data*)

Pass some received HTTP/2 data to the connection for handling.

Parameters **data** (bytes) – The data received from the remote peer on the network.

Returns A list of events that the remote peer triggered by sending this data.

remote_flow_control_window(*stream_id*)

Returns the maximum amount of data the remote peer can send on stream stream_id.

This value will never be larger than the total data that can be sent on the connection: even if the given stream allows more data, the connection window provides a logical maximum to the amount of data that can be sent.

The maximum data that can be sent in a single data frame on a stream is either this value, or the maximum frame size, whichever is *smaller*.

Parameters **stream_id** (int) – The ID of the stream whose flow control window is being queried.

Returns The amount of data in bytes that can be received on the stream before the flow control window is exhausted.

Return type int

reset_stream(*stream_id*, *error_code=0*)

Reset a stream.

This method forcibly closes a stream by sending a RST_STREAM frame for a given stream. This is not a graceful closure. To gracefully end a stream, try the *end_stream* method.

Parameters

- **stream_id** (int) – The ID of the stream to reset.
- **error_code** (int) – (optional) The error code to use to reset the stream. Defaults to *ErrorCodes.NO_ERROR*.

Returns Nothing

send_data(*stream_id*, *data*, *end_stream=False*, *pad_length=None*)

Send data on a given stream.

This method does no breaking up of data: if the data is larger than the value returned by *local_flow_control_window* for this stream then a *FlowControlError* will be raised. If the data is larger than *max_outbound_frame_size* then a *FrameTooLargeError* will be raised.

h2 does this to avoid buffering the data internally. If the user has more data to send than h2 will allow, consider breaking it up and buffering it externally.

Parameters

- **stream_id** (int) – The ID of the stream on which to send the data.

- **data** (bytes) – The data to send on the stream.
- **end_stream** (bool) – (optional) Whether this is the last data to be sent on the stream. Defaults to False.
- **pad_length** (int) – (optional) Length of the padding to apply to the data frame. Defaults to None for no use of padding. Note that a value of 0 results in padding of length 0 (with the "padding" flag set on the frame).

  New in version 2.6.0.

Returns Nothing

send_headers(*stream_id*, *headers*, *end_stream=False*, *priority_weight=None*, *priority_depends_on=None*, *priority_exclusive=None*)

Send headers on a given stream.

This function can be used to send request or response headers: the kind that are sent depends on whether this connection has been opened as a client or server connection, and whether the stream was opened by the remote peer or not.

If this is a client connection, calling `send_headers` will send the headers as a request. It will also implicitly open the stream being used. If this is a client connection and `send_headers` has *already* been called, this will send trailers instead.

If this is a server connection, calling `send_headers` will send the headers as a response. It is a protocol error for a server to open a stream by sending headers. If this is a server connection and `send_headers` has *already* been called, this will send trailers instead.

When acting as a server, you may call `send_headers` any number of times allowed by the following rules, in this order:

- zero or more times with `(':status', '1XX')` (where `1XX` is a placeholder for any 100-level status code).
- once with any other status header.
- zero or one time for trailers.

That is, you are allowed to send as many informational responses as you like, followed by one complete response and zero or one HTTP trailer blocks.

Clients may send one or two header blocks: one request block, and optionally one trailer block.

If it is important to send HPACK "never indexed" header fields (as defined in RFC 7451 Section 7.1.3), the user may instead provide headers using the HPACK library's `HeaderTuple` and `NeverIndexedHeaderTuple` objects.

This method also allows users to prioritize the stream immediately, by sending priority information on the HEADERS frame directly. To do this, any one of `priority_weight`, `priority_depends_on`, or `priority_exclusive` must be set to a value that is not `None`. For more information on the priority fields, see *`prioritize`*.

> **Warning:** In HTTP/2, it is mandatory that all the HTTP/2 special headers (that is, ones whose header keys begin with `:`) appear at the start of the header block, before any normal headers.

Changed in version 2.3.0: Added support for using `HeaderTuple` objects to store headers.

Changed in version 2.4.0: Added the ability to provide priority keyword arguments: `priority_weight`, `priority_depends_on`, and `priority_exclusive`.

Parameters

- **stream_id** (int) – The stream ID to send the headers on. If this stream does not currently exist, it will be created.
- **headers** (An iterable of two tuples of bytestrings or HeaderTuple objects.) – The request/response headers to send.
- **end_stream** (bool) – Whether this headers frame should end the stream immediately (that is, whether no more data will be sent after this frame). Defaults to False.
- **priority_weight** (int or None) – Sets the priority weight of the stream. See *prioritize* for more about how this field works. Defaults to None, which means that no priority information will be sent.
- **priority_depends_on** (bool or None) – Sets which stream this one depends on for priority purposes. See *prioritize* for more about how this field works. Defaults to None, which means that no priority information will be sent.
- **priority_exclusive** – Sets whether this stream exclusively depends on the stream given in priority_depends_on for priority purposes. See *prioritize* for more about how this field workds. Defaults to None, which means that no priority information will be sent.

Returns Nothing

update_settings(*new_settings*)

Update the local settings. This will prepare and emit the appropriate SETTINGS frame.

Parameters new_settings – A dictionary of {setting: new value}

1.7.3 Configuration

class h2.config.**H2Configuration**(*client_side=True*, *header_encoding=None*, *validate_outbound_headers=True*, *normalize_outbound_headers=True*, *validate_inbound_headers=True*, *normalize_inbound_headers=True*, *logger=None*)

An object that controls the way a single HTTP/2 connection behaves.

This object allows the users to customize behaviour. In particular, it allows users to enable or disable optional features, or to otherwise handle various unusual behaviours.

This object has very little behaviour of its own: it mostly just ensures that configuration is self-consistent.

Parameters

- **client_side** (bool) – Whether this object is to be used on the client side of a connection, or on the server side. Affects the logic used by the state machine, the default settings values, the allowable stream IDs, and several other properties. Defaults to True.
- **header_encoding** (str, False, or None) – Controls whether the headers emitted by this object in events are transparently decoded to unicode strings, and what encoding is used to do that decoding. This defaults to None, meaning that headers will be returned as bytes. To automatically decode headers (that is, to return them as unicode strings), this can be set to the string name of any encoding, e.g. 'utf-8'.

  Changed in version 3.0.0: Changed default value from 'utf-8' to None
- **validate_outbound_headers** (bool) – Controls whether the headers emitted by this object are validated against the rules in RFC 7540. Disabling this setting will cause outbound header validation to be skipped, and allow the object to emit headers that may be illegal according to RFC 7540. Defaults to True.

- **normalize_outbound_headers** (bool) – Controls whether the headers emitted by this object are normalized before sending. Disabling this setting will cause outbound header normalization to be skipped, and allow the object to emit headers that may be illegal according to RFC 7540. Defaults to True.
- **validate_inbound_headers** (bool) – Controls whether the headers received by this object are validated against the rules in RFC 7540. Disabling this setting will cause inbound header validation to be skipped, and allow the object to receive headers that may be illegal according to RFC 7540. Defaults to True.
- **normalize_inbound_headers** (bool) – Controls whether the headers received by this object are normalized according to the rules of RFC 7540. Disabling this setting may lead to h2 emitting header blocks that some RFCs forbid, e.g. with multiple cookie fields.

  New in version 3.0.0.
- **logger** (logging.Logger) – A logger that conforms to the requirements for this module, those being no I/O and no context switches, which is needed in order to run in asynchronous operation.

  New in version 2.6.0.

header_encoding

Controls whether the headers emitted by this object in events are transparently decoded to unicode strings, and what encoding is used to do that decoding. This defaults to None, meaning that headers will be returned as bytes. To automatically decode headers (that is, to return them as unicode strings), this can be set to the string name of any encoding, e.g. 'utf-8'.

1.7.4 Events

class h2.events.**RequestReceived**

The RequestReceived event is fired whenever request headers are received. This event carries the HTTP headers for the given request and the stream ID of the new stream.

Changed in version 2.3.0: Changed the type of headers to HeaderTuple. This has no effect on current users.

Changed in version 2.4.0: Added stream_ended and priority_updated properties.

headers = None

The request headers.

priority_updated = None

If this request also had associated priority information, the associated *PriorityUpdated* event will be available here.

New in version 2.4.0.

stream_ended = None

If this request also ended the stream, the associated *StreamEnded* event will be available here.

New in version 2.4.0.

stream_id = None

The Stream ID for the stream this request was made on.

class h2.events.**ResponseReceived**

The ResponseReceived event is fired whenever response headers are received. This event carries the HTTP headers for the given response and the stream ID of the new stream.

Changed in version 2.3.0: Changed the type of `headers` to `HeaderTuple`. This has no effect on current users.

Changed in version 2.4.0: Added `stream_ended` and `priority_updated` properties.

headers = None
The response headers.

priority_updated = None
If this response also had associated priority information, the associated *PriorityUpdated* event will be available here.

New in version 2.4.0.

stream_ended = None
If this response also ended the stream, the associated *StreamEnded* event will be available here.

New in version 2.4.0.

stream_id = None
The Stream ID for the stream this response was made on.

class h2.events.**TrailersReceived**
The TrailersReceived event is fired whenever trailers are received on a stream. Trailers are a set of headers sent after the body of the request/response, and are used to provide information that wasn't known ahead of time (e.g. content-length). This event carries the HTTP header fields that form the trailers and the stream ID of the stream on which they were received.

Changed in version 2.3.0: Changed the type of `headers` to `HeaderTuple`. This has no effect on current users.

Changed in version 2.4.0: Added `stream_ended` and `priority_updated` properties.

headers = None
The trailers themselves.

priority_updated = None
If the trailers also set associated priority information, the associated *PriorityUpdated* event will be available here.

New in version 2.4.0.

stream_ended = None
Trailers always end streams. This property has the associated *StreamEnded* in it.

New in version 2.4.0.

stream_id = None
The Stream ID for the stream on which these trailers were received.

class h2.events.**InformationalResponseReceived**
The InformationalResponseReceived event is fired when an informational response (that is, one whose status code is a 1XX code) is received from the remote peer.

The remote peer may send any number of these, from zero upwards. These responses are most commonly sent in response to requests that have the `expect: 100-continue` header field present. Most users can safely ignore this event unless you are intending to use the `expect: 100-continue` flow, or are for any reason expecting a different 1XX status code.

New in version 2.2.0.

Changed in version 2.3.0: Changed the type of `headers` to `HeaderTuple`. This has no effect on current users.

Changed in version 2.4.0: Added `priority_updated` property.

headers = None

The headers for this informational response.

priority_updated = None

If this response also had associated priority information, the associated *PriorityUpdated* event will be available here.

New in version 2.4.0.

stream_id = None

The Stream ID for the stream this informational response was made on.

class h2.events.**DataReceived**

The DataReceived event is fired whenever data is received on a stream from the remote peer. The event carries the data itself, and the stream ID on which the data was received.

Changed in version 2.4.0: Added `stream_ended` property.

data = None

The data itself.

flow_controlled_length = None

The amount of data received that counts against the flow control window. Note that padding counts against the flow control window, so when adjusting flow control you should always use this field rather than `len(data)`.

stream_ended = None

If this data chunk also completed the stream, the associated *StreamEnded* event will be available here.

New in version 2.4.0.

stream_id = None

The Stream ID for the stream this data was received on.

class h2.events.**WindowUpdated**

The WindowUpdated event is fired whenever a flow control window changes size. HTTP/2 defines flow control windows for connections and streams: this event fires for both connections and streams. The event carries the ID of the stream to which it applies (set to zero if the window update applies to the connection), and the delta in the window size.

delta = None

The window delta.

stream_id = None

The Stream ID of the stream whose flow control window was changed. May be `0` if the connection window was changed.

class h2.events.**RemoteSettingsChanged**

The RemoteSettingsChanged event is fired whenever the remote peer changes its settings. It contains a complete inventory of changed settings, including their previous values.

In HTTP/2, settings changes need to be acknowledged. h2 automatically acknowledges settings changes for efficiency. However, it is possible that the caller may not be happy with the changed setting.

When this event is received, the caller should confirm that the new settings are acceptable. If they are not acceptable, the user should close the connection with the error code *PROTOCOL_ERROR*.

Changed in version 2.0.0: Prior to this version the user needed to acknowledge settings changes. This is no longer the case: h2 now automatically acknowledges them.

changed_settings = None

A dictionary of setting byte to *ChangedSetting*, representing the changed settings.

classmethod from_settings(*old_settings*, *new_settings*)

Build a RemoteSettingsChanged event from a set of changed settings.

Parameters

- **old_settings** – A complete collection of old settings, in the form of a dictionary of `{setting: value}`.
- **new_settings** – All the changed settings and their new values, in the form of a dictionary of `{setting: value}`.

class h2.events.**PingReceived**

The PingReceived event is fired whenever a PING is received. It contains the 'opaque data' of the PING frame. A ping acknowledgment with the same 'opaque data' is automatically emitted after receiving a ping.

New in version 3.1.0.

ping_data = None

The data included on the ping.

class h2.events.**PingAckReceived**

The PingAckReceived event is fired whenever a PING acknowledgment is received. It contains the 'opaque data' of the PING+ACK frame, allowing the user to correlate PINGs and calculate RTT.

New in version 3.1.0.

Changed in version 4.0.0: Removed deprecated but equivalent `PingAcknowledged`.

ping_data = None

The data included on the ping.

class h2.events.**StreamEnded**

The StreamEnded event is fired whenever a stream is ended by a remote party. The stream may not be fully closed if it has not been closed locally, but no further data or headers should be expected on that stream.

stream_id = None

The Stream ID of the stream that was closed.

class h2.events.**StreamReset**

The StreamReset event is fired in two situations. The first is when the remote party forcefully resets the stream. The second is when the remote party has made a protocol error which only affects a single stream. In this case, h2 will terminate the stream early and return this event.

Changed in version 2.0.0: This event is now fired when h2 automatically resets a stream.

error_code = None

The error code given. Either one of *ErrorCodes* or `int`

remote_reset = None

Whether the remote peer sent a RST_STREAM or we did.

stream_id = None

The Stream ID of the stream that was reset.

class h2.events.**PushedStreamReceived**

The PushedStreamReceived event is fired whenever a pushed stream has been received from a remote peer. The event carries on it the new stream ID, the ID of the parent stream, and the request headers pushed by the remote peer.

headers = None

The request headers, sent by the remote party in the push.

parent_stream_id = None
The Stream ID of the stream that the push is related to.

pushed_stream_id = None
The Stream ID of the stream created by the push.

class h2.events.**SettingsAcknowledged**
The SettingsAcknowledged event is fired whenever a settings ACK is received from the remote peer. The event carries on it the settings that were acknowedged, in the same format as *h2.events.RemoteSettingsChanged*.

changed_settings = None
A dictionary of setting byte to *ChangedSetting*, representing the changed settings.

class h2.events.**PriorityUpdated**
The PriorityUpdated event is fired whenever a stream sends updated priority information. This can occur when the stream is opened, or at any time during the stream lifetime.

This event is purely advisory, and does not need to be acted on.

New in version 2.0.0.

depends_on = None
The stream ID this stream now depends on. May be `0`.

exclusive = None
Whether the stream *exclusively* depends on the parent stream. If it does, this stream should inherit the current children of its new parent.

stream_id = None
The ID of the stream whose priority information is being updated.

weight = None
The new stream weight. May be the same as the original stream weight. An integer between 1 and 256.

class h2.events.**ConnectionTerminated**
The ConnectionTerminated event is fired when a connection is torn down by the remote peer using a GOAWAY frame. Once received, no further action may be taken on the connection: a new connection must be established.

additional_data = None
Additional debug data that can be appended to GOAWAY frame.

error_code = None
The error code cited when tearing down the connection. Should be one of *ErrorCodes*, but may not be if unknown HTTP/2 extensions are being used.

last_stream_id = None
The stream ID of the last stream the remote peer saw. This can provide an indication of what data, if any, never reached the remote peer and so can safely be resent.

class h2.events.**AlternativeServiceAvailable**
The AlternativeServiceAvailable event is fired when the remote peer advertises an RFC 7838 Alternative Service using an ALTSVC frame.

This event always carries the origin to which the ALTSVC information applies. That origin is either supplied by the server directly, or inferred by h2 from the `:authority` pseudo-header field that was sent by the user when initiating a given stream.

This event also carries what RFC 7838 calls the “Alternative Service Field Value”, which is formatted like a HTTP header field and contains the relevant alternative service information. h2 does not parse or in any way modify that information: the user is required to do that.

This event can only be fired on the client end of a connection.

New in version 2.3.0.

field_value = None

The ALTSVC field value. This contains information about the HTTP alternative service being advertised by the server. h2 does not parse this field: it is left exactly as sent by the server. The structure of the data in this field is given by RFC 7838 Section 3.

origin = None

The origin to which the alternative service field value applies. This field is either supplied by the server directly, or inferred by h2 from the `:authority` pseudo-header field that was sent by the user when initiating the stream on which the frame was received.

class `h2.events.`**UnknownFrameReceived**

The UnknownFrameReceived event is fired when the remote peer sends a frame that h2 does not understand. This occurs primarily when the remote peer is employing HTTP/2 extensions that h2 doesn't know anything about.

RFC 7540 requires that HTTP/2 implementations ignore these frames. h2 does so. However, this event is fired to allow implementations to perform special processing on those frames if needed (e.g. if the implementation is capable of handling the frame itself).

New in version 2.7.0.

frame = None

The hyperframe Frame object that encapsulates the received frame.

1.7.5 Exceptions

class `h2.exceptions.`**H2Error**

The base class for all exceptions for the HTTP/2 module.

class `h2.exceptions.`**NoSuchStreamError**(*stream_id*)

Bases: *h2.exceptions.ProtocolError*

A stream-specific action referenced a stream that does not exist.

Changed in version 2.0.0: Became a subclass of *ProtocolError*

stream_id = None

The stream ID corresponds to the non-existent stream.

class `h2.exceptions.`**StreamClosedError**(*stream_id*)

Bases: *h2.exceptions.NoSuchStreamError*

A more specific form of *NoSuchStreamError*. Indicates that the stream has since been closed, and that all state relating to that stream has been removed.

stream_id = None

The stream ID corresponds to the nonexistent stream.

class `h2.exceptions.`**RFC1122Error**

Bases: *h2.exceptions.H2Error*

Emitted when users attempt to do something that is literally allowed by the relevant RFC, but is sufficiently ill-defined that it's unwise to allow users to actually do it.

While there is some disagreement about whether or not we should be liberal in what accept, it is a truth universally acknowledged that we should be conservative in what emit.

New in version 2.4.0.

Protocol Errors

class h2.exceptions.**ProtocolError**

Bases: *h2.exceptions.H2Error*

An action was attempted in violation of the HTTP/2 protocol.

error_code = 1

The error code corresponds to this kind of Protocol Error.

class h2.exceptions.**FrameTooLargeError**

Bases: *h2.exceptions.ProtocolError*

The frame that we tried to send or that we received was too large.

error_code = 6

The error code corresponds to this kind of Protocol Error.

class h2.exceptions.**FrameDataMissingError**

Bases: *h2.exceptions.ProtocolError*

The frame that we received is missing some data.

New in version 2.0.0.

error_code = 6

The error code corresponds to this kind of Protocol Error.

class h2.exceptions.**TooManyStreamsError**

Bases: *h2.exceptions.ProtocolError*

An attempt was made to open a stream that would lead to too many concurrent streams.

class h2.exceptions.**FlowControlError**

Bases: *h2.exceptions.ProtocolError*

An attempted action violates flow control constraints.

error_code = 3

The error code corresponds to this kind of Protocol Error.

class h2.exceptions.**StreamIDTooLowError**(*stream_id*, *max_stream_id*)

Bases: *h2.exceptions.ProtocolError*

An attempt was made to open a stream that had an ID that is lower than the highest ID we have seen on this connection.

max_stream_id = None

The current highest-seen stream ID.

stream_id = None

The ID of the stream that we attempted to open.

class h2.exceptions.**InvalidSettingsValueError**(*msg*, *error_code*)

An attempt was made to set an invalid Settings value.

New in version 2.0.0.

class h2.exceptions.**NoAvailableStreamIDError**

Bases: *h2.exceptions.ProtocolError*

There are no available stream IDs left to the connection. All stream IDs have been exhausted.

New in version 2.0.0.

class h2.exceptions.**InvalidBodyLengthError**(*expected*, *actual*)

Bases: *h2.exceptions.ProtocolError*

The remote peer sent more or less data that the Content-Length header indicated.

New in version 2.0.0.

class h2.exceptions.**UnsupportedFrameError**

The remote peer sent a frame that is unsupported in this context.

New in version 2.1.0.

Changed in version 4.0.0: Removed deprecated KeyError parent class.

class h2.exceptions.**DenialOfServiceError**

Bases: *h2.exceptions.ProtocolError*

Emitted when the remote peer exhibits a behaviour that is likely to be an attempt to perform a Denial of Service attack on the implementation. This is a form of ProtocolError that carries a different error code, and allows more easy detection of this kind of behaviour.

New in version 2.5.0.

error_code = 11

The error code corresponds to this kind of *ProtocolError*

1.7.6 HTTP/2 Error Codes

h2/errors

Global error code registry containing the established HTTP/2 error codes.

The current registry is available at: https://tools.ietf.org/html/rfc7540#section-11.4

class h2.errors.**ErrorCodes**

All known HTTP/2 error codes.

New in version 2.5.0.

CANCEL = 8

Stream cancelled.

COMPRESSION_ERROR = 9

Compression state not updated.

CONNECT_ERROR = 10

TCP connection error for CONNECT method.

ENHANCE_YOUR_CALM = 11

Processing capacity exceeded.

FLOW_CONTROL_ERROR = 3

Flow-control limits exceeded.

FRAME_SIZE_ERROR = 6

Frame size incorrect.

HTTP_1_1_REQUIRED = 13

Use HTTP/1.1 for the request.

INADEQUATE_SECURITY = 12

Negotiated TLS parameters not acceptable.

INTERNAL_ERROR = 2
Implementation fault.

NO_ERROR = 0
Graceful shutdown.

PROTOCOL_ERROR = 1
Protocol error detected.

REFUSED_STREAM = 7
Stream not processed.

SETTINGS_TIMEOUT = 4
Settings not acknowledged.

STREAM_CLOSED = 5
Frame received for closed stream.

1.7.7 Settings

class h2.settings.**SettingCodes**
All known HTTP/2 setting codes.

New in version 2.6.0.

ENABLE_CONNECT_PROTOCOL = 8
This setting can be used to enable the connect protocol. To enable on a client set this to 1.

ENABLE_PUSH = 2
This setting can be used to disable server push. To disable server push on a client, set this to 0.

HEADER_TABLE_SIZE = 1
Allows the sender to inform the remote endpoint of the maximum size of the header compression table used to decode header blocks, in octets.

INITIAL_WINDOW_SIZE = 4
Indicates the sender's initial window size (in octets) for stream-level flow control.

MAX_CONCURRENT_STREAMS = 3
Indicates the maximum number of concurrent streams that the sender will allow.

MAX_FRAME_SIZE = 5
Indicates the size of the largest frame payload that the sender is willing to receive, in octets.

MAX_HEADER_LIST_SIZE = 6
This advisory setting informs a peer of the maximum size of header list that the sender is prepared to accept, in octets. The value is based on the uncompressed size of header fields, including the length of the name and value in octets plus an overhead of 32 octets for each header field.

class h2.settings.**Settings**(*client=True*, *initial_values=None*)
An object that encapsulates HTTP/2 settings state.

HTTP/2 Settings are a complex beast. Each party, remote and local, has its own settings and a view of the other party's settings. When a settings frame is emitted by a peer it cannot assume that the new settings values are in place until the remote peer acknowledges the setting. In principle, multiple settings changes can be "in flight" at the same time, all with different values.

This object encapsulates this mess. It provides a dict-like interface to settings, which return the *current* values of the settings in question. Additionally, it keeps track of the stack of proposed values: each time an acknowledgement is sent/received, it updates the current values with the stack of proposed values. On top of all that, it validates the values to make sure they're allowed, and raises *InvalidSettingsValueError* if they are not.

Finally, this object understands what the default values of the HTTP/2 settings are, and sets those defaults appropriately.

Changed in version 2.2.0: Added the `initial_values` parameter.

Changed in version 2.5.0: Added the `max_header_list_size` property.

Parameters

- **client** (`bool`) – (optional) Whether these settings should be defaulted for a client implementation or a server implementation. Defaults to `True`.
- **initial_values** – (optional) Any initial values the user would like set, rather than RFC 7540's defaults.

acknowledge()

The settings have been acknowledged, either by the user (remote settings) or by the remote peer (local settings).

Returns A dict of {setting: ChangedSetting} that were applied.

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

enable_connect_protocol

The current value of the *ENABLE_CONNECT_PROTOCOL* setting.

enable_push

The current value of the *ENABLE_PUSH* setting.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

header_table_size

The current value of the *HEADER_TABLE_SIZE* setting.

initial_window_size

The current value of the *INITIAL_WINDOW_SIZE* setting.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

max_concurrent_streams

The current value of the *MAX_CONCURRENT_STREAMS* setting.

max_frame_size

The current value of the *MAX_FRAME_SIZE* setting.

max_header_list_size

The current value of the *MAX_HEADER_LIST_SIZE* setting. If not set, returns `None`, which means unlimited.

New in version 2.5.0.

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() → an object providing a view on D's values

class h2.settings.**ChangedSetting**(*setting*, *original_value*, *new_value*)

new_value = None
The new value after being changed.

original_value = None
The original value before being changed.

setting = None
The setting code given. Either one of *SettingCodes* or int

Changed in version 2.6.0.

1.8 Testimonials

1.8.1 Glyph Lefkowitz

Frankly, Hyper-h2 is almost SURREAL in how well-factored and decoupled the implementation is from I/O. If libraries in the Python ecosystem looked like this generally, Twisted would be a much better platform than it is. (Frankly, most of Twisted's _own_ protocol implementations should aspire to such cleanliness.)

(Source)

1.9 Release Process

Because of h2's place at the bottom of the dependency tree, it is extremely important that the project maintains a diligent release schedule. This document outlines our process for managing releases.

1.9.1 Versioning

h2 follows semantic versioning of its public API when it comes to numbering releases. The public API of h2 is strictly limited to the entities listed in the *h2 API* documentation: anything not mentioned in that document is not considered part of the public API and is not covered by the versioning guarantees given by semantic versioning.

1.9.2 Maintenance

h2 has the notion of a "release series", given by a major and minor version number: for example, there is the 2.1 release series. When each minor release is made and a release series is born, a branch is made off the release tag: for example, for the 2.1 release series, the 2.1.X branch.

All changes merged into the master branch will be evaluated for whether they can be considered 'bugfixes' only (that is, they do not affect the public API). If they can, they will also be cherry-picked back to all active maintenance branches that require the bugfix. If the bugfix is not necessary, because the branch in question is unaffected by that bug, the bugfix will not be backported.

1.9.3 Supported Release Series'

The developers of h2 commit to supporting the following release series:

- The most recent, as identified by the first two numbers in the highest version currently released.
- The immediately prior release series.

The only exception to this policy is that no release series earlier than the 2.1 series will be supported. In this context, "supported" means that they will continue to receive bugfix releases.

For releases other than the ones identified above, no support is guaranteed. The developers may *choose* to support such a release series, but they do not promise to.

The exception here is for security vulnerabilities. If a security vulnerability is identified in an out-of-support release series, the developers will do their best to patch it and issue an emergency release. For more information, see our security documentation.

1.10 Release Notes

This document contains release notes for Hyper-h2. In addition to the *Release History* found at the bottom of this document, this document also includes a high-level prose overview of each major release after 1.0.0.

1.10.1 High Level Notes

3.0.0: 24 March 2017

The Hyper-h2 team and the Hyper project are delighted to announce the release of Hyper-h2 version 3.0.0! Unlike the really notable 2.0.0 release, this release is proportionally quite small: however, it has the effect of removing a lot of cruft and complexity that has built up in the codebase over the lifetime of the v2 release series.

This release was motivated primarily by discovering that applications that attempted to use both HTTP/1.1 and HTTP/2 using hyper-h2 would encounter problems with cookies, because hyper-h2 did not join together cookie headers as required by RFC 7540. Normally adding such behaviour would be a non-breaking change, but we previously had no flags to prevent normalization of received HTTP headers.

Because it makes no sense for the cookie to be split *by default*, we needed to add a controlling flag and set it to true. The breaking nature of this change is very subtle, and it's possible most users would never notice, but nevertheless it *is* a breaking change and we need to treat it as such.

Happily, we can take this opportunity to finalise a bunch of deprecations we'd made over the past year. The v2 release series was long-lived and successful, having had a series of releases across the past year-and-a-bit, and the Hyper team are very proud of it. However, it's time to open a new chapter, and remove the deprecated code.

The past year has been enormously productive for the Hyper team. A total of 30 v2 releases were made, an enormous amount of work. A good number of people have made their first contribution in this time, more than I can thank reasonably without taking up an unreasonable amount of space in this document, so instead I invite you to check out our awesome contributor list.

We're looking forward to the next chapter in hyper-h2: it's been a fun ride so far, and we hope even more of you come along and join in the fun over the next year!

2.0.0: 25 January 2016

The Hyper-h2 team and the Hyper project are delighted to announce the release of Hyper-h2 version 2.0.0! This is an enormous release that contains a gigantic collection of new features and fixes, with the goal of making it easier than ever to use Hyper-h2 to build a compliant HTTP/2 server or client.

An enormous chunk of this work has been focused on tighter enforcement of restrictions in RFC 7540, ensuring that we correctly police the actions of remote peers, and error appropriately when those peers violate the specification. Several of these constitute breaking changes, because data that was previously received and handled without obvious error now raises `ProtocolError` exceptions and causes the connection to be terminated.

Additionally, the public API was cleaned up and had several helper methods that had been inadvertently exposed removed from the public API. The team wants to stress that while Hyper-h2 follows semantic versioning, the guarantees of semver apply only to the public API as documented in *h2 API*. Reducing the surface area of these APIs makes it easier for us to continue to ensure that the guarantees of semver are respected on our public API.

We also attempted to clear up some of the warts that had appeared in the API, and add features that are helpful for implementing HTTP/2 endpoints. For example, the `H2Connection` object now exposes a method for generating the next stream ID that your client or server can use to initiate a connection (`get_next_available_stream_id`). We also removed some needless return values that were guaranteed to return empty lists, which were an attempt to make a forward-looking guarantee that was entirely unneeded.

Altogether, this has been an extremely productive period for Hyper-h2, and a lot of great work has been done by the community. To that end, we'd also like to extend a great thankyou to those contributors who made their first contribution to the project between release 1.0.0 and 2.0.0. Many thanks to: Thomas Kriechbaumer, Alex Chan, Maximilian Hils, and Glyph. For a full historical list of contributors, see contributors.

We're looking forward to the next few months of Python HTTP/2 work, and hoping that you'll find lots of excellent HTTP/2 applications to build with Hyper-h2!

1.11 Release History

1.11.1 4.1.0 (2021-10-05)

Note: The GitHub repository has been renamed to `python-hyper/h2`, previously was `python-hyper/hyper-h2`. **The name of the package on PyPI is unchanged!**

API Changes (Backward-Compatible)

- Support for Python 3.9 has been added.
- Support for Python 3.10 has been added.
- New example for a Python socket HTTP/2 client.
- New *OutputLogger* for use with `h2.config.logger`. This is only provided for convenience and not part of the stable API.

Bugfixes

- Header validation now rejects empty header names with a ProtocolError. While hpack decodes such header blocks without issues, they violate the HTTP semantics.
- Fix TE header name in error message.

1.11.2 4.0.0 (2020-09-19)

API Changes (Backward-Incompatible)

- Support for Python 2.7 has been removed.
- Support for Python 3.4 has been removed.
- Support for Python 3.5 has been removed.
- Support for PyPy (Python 2.7 compatible) has been removed.
- Support for Python 3.8 has been added.
- Receiving DATA before HEADERS now raises a ProtocolError (see https://tools.ietf.org/html/rfc7540#section-8.1)

1.11.3 3.2.0 (2020-02-08)

Bugfixes

- Receiving DATA frames on closed (or reset) streams now properly emit a WINDOW_UPDATE to keep the connection flow window topped up.

API Changes (Backward-Incompatible)

- `h2.config.logger` now uses a *trace(. . .)* function, in addition to *debug(. . .)*. If you defined a custom logger object, you need to handle these new function calls.

1.11.4 3.1.1 (2019-08-02)

Bugfixes

- Ignore WINDOW_UPDATE and RST_STREAM frames received after stream closure.

1.11.5 3.1.0 (2019-01-22)

API Changes (Backward-Incompatible)

- `h2.connection.H2Connection.data_to_send` first and only argument `amt` was renamed to `amount`.
- Support for Python 3.3 has been removed.

API Changes (Backward-Compatible)

- `h2.connection.H2Connection.send_data` now supports `data` parameter being a `memoryview` object.
- Refactor ping-related events: a `h2.events.PingReceived` event is fired when a PING frame is received and a `h2.events.PingAckReceived` event is fired when a PING frame with an ACK flag is received. `h2.events.PingAcknowledged` is deprecated in favour of the identical `h2.events.PingAckReceived`.

- Added `ENABLE_CONNECT_PROTOCOL` to `h2.settings.SettingCodes`.
- Support `CONNECT` requests with a `:protocol` pseudo header thereby supporting RFC 8441.
- A limit to the number of closed streams kept in memory by the connection is applied. It can be configured by `h2.connection.H2Connection.MAX_CLOSED_STREAMS`.

Bugfixes

- Debug logging when stream_id is None is now fixed and no longer errors.

1.11.6 3.0.1 (2017-04-03)

Bugfixes

- CONTINUATION frames sent on closed streams previously caused stream errors of type STREAM_CLOSED. RFC 7540 § 6.10 requires that these be connection errors of type PROTOCOL_ERROR, and so this release changes to match that behaviour.
- Remote peers incrementing their inbound connection window beyond the maximum allowed value now cause stream-level errors, rather than connection-level errors, allowing connections to stay up longer.
- h2 now rejects receiving and sending request header blocks that are missing any of the mandatory pseudo-header fields (:path, :scheme, and :method).
- h2 now rejects receiving and sending request header blocks that have an empty :path pseudo-header value.
- h2 now rejects receiving and sending request header blocks that contain response-only pseudo-headers, and vice versa.
- h2 now correct respects user-initiated changes to the HEADER_TABLE_SIZE local setting, and ensures that if users shrink or increase the header table size it is policed appropriately.

1.11.7 2.6.2 (2017-04-03)

Bugfixes

- CONTINUATION frames sent on closed streams previously caused stream errors of type STREAM_CLOSED. RFC 7540 § 6.10 requires that these be connection errors of type PROTOCOL_ERROR, and so this release changes to match that behaviour.
- Remote peers incrementing their inbound connection window beyond the maximum allowed value now cause stream-level errors, rather than connection-level errors, allowing connections to stay up longer.
- h2 now rejects receiving and sending request header blocks that are missing any of the mandatory pseudo-header fields (:path, :scheme, and :method).
- h2 now rejects receiving and sending request header blocks that have an empty :path pseudo-header value.
- h2 now rejects receiving and sending request header blocks that contain response-only pseudo-headers, and vice versa.
- h2 now correct respects user-initiated changes to the HEADER_TABLE_SIZE local setting, and ensures that if users shrink or increase the header table size it is policed appropriately.

1.11.8 2.5.4 (2017-04-03)

Bugfixes

- CONTINUATION frames sent on closed streams previously caused stream errors of type STREAM_CLOSED. RFC 7540 § 6.10 requires that these be connection errors of type PROTOCOL_ERROR, and so this release changes to match that behaviour.
- Remote peers incrementing their inbound connection window beyond the maximum allowed value now cause stream-level errors, rather than connection-level errors, allowing connections to stay up longer.
- h2 now correct respects user-initiated changes to the HEADER_TABLE_SIZE local setting, and ensures that if users shrink or increase the header table size it is policed appropriately.

1.11.9 3.0.0 (2017-03-24)

API Changes (Backward-Incompatible)

- By default, hyper-h2 now joins together received cookie header fields, per RFC 7540 Section 8.1.2.5.
- Added a `normalize_inbound_headers` flag to the `H2Configuration` object that defaults to `True`. Setting this to `False` changes the behaviour from the previous point back to the v2 behaviour.
- Removed deprecated fields from `h2.errors` module.
- Removed deprecated fields from `h2.settings` module.
- Removed deprecated `client_side` and `header_encoding` arguments from `H2Connection`.
- Removed deprecated `client_side` and `header_encoding` properties from `H2Connection`.
- `dict` objects are no longer allowed for user-supplied headers.
- The default header encoding is now `None`, not `utf-8`: this means that all events that carry headers now return those headers as byte strings by default. The header encoding can be set back to `utf-8` to restore the old behaviour.

API Changes (Backward-Compatible)

- Added new `UnknownFrameReceived` event that fires when unknown extension frames have been received. This only fires when using hyperframe 5.0 or later: earlier versions of hyperframe cause us to silently ignore extension frames.

Bugfixes

None

1.11.10 2.6.1 (2017-03-16)

Bugfixes

- Allowed hyperframe v5 support while continuing to ignore unexpected frames.

1.11.11 2.5.3 (2017-03-16)

Bugfixes

- Allowed hyperframe v5 support while continuing to ignore unexpected frames.

1.11.12 2.4.4 (2017-03-16)

Bugfixes

- Allowed hyperframe v5 support while continuing to ignore unexpected frames.

1.11.13 2.6.0 (2017-02-28)

API Changes (Backward-Compatible)

- Added a new `h2.events.Event` class that acts as a base class for all events.
- Rather than reject outbound Connection-specific headers, h2 will now normalize the header block by removing them.
- Implement equality for the `h2.settings.Settings` class.
- Added `h2.settings.SettingCodes`, an enum that is used to store all the HTTP/2 setting codes. This allows us to use a better printed representation of the setting code in most places that it is used.
- The `setting` field in `ChangedSetting` for the `RemoteSettingsChanged` and `SettingsAcknowledged` events has been updated to be instances of `SettingCodes` whenever they correspond to a known setting code. When they are an unknown setting code, they are instead `int`. As `SettingCodes` is a subclass of `int`, this is non-breaking.
- Deprecated the other fields in `h2.settings`. These will be removed in 3.0.0.
- Added an optional `pad_length` parameter to `H2Connection.send_data` to allow the user to include padding on a data frame.
- Added a new parameter to the `h2.config.H2Configuration` initializer which takes a logger. This allows us to log by providing a logger that conforms to the requirements of this module so that it can be used in different environments.

Bugfixes

- Correctly reject pushed request header blocks whenever they have malformed request header blocks.
- Correctly normalize pushed request header blocks whenever they have normalizable header fields.
- Remote peers are now allowed to send zero or any positive number as a value for `SETTINGS_MAX_HEADER_LIST_SIZE`, where previously sending zero would raise a `InvalidSettingsValueError`.
- Resolved issue where the `HTTP2-Settings` header value for plaintext upgrade that was emitted by `initiate_upgrade_connection` included the *entire* `SETTINGS` frame, instead of just the payload.
- Resolved issue where the `HTTP2-Settings` header value sent by a client for plaintext upgrade would be ignored by `initiate_upgrade_connection`, rather than have those settings applied appropriately.

- Resolved an issue whereby certain frames received from a peer in the CLOSED state would trigger connection errors when RFC 7540 says they should have triggered stream errors instead. Added more detailed stream closure tracking to ensure we don't throw away connections unnecessarily.

1.11.14 2.5.2 (2017-01-27)

- Resolved issue where the `HTTP2-Settings` header value for plaintext upgrade that was emitted by `initiate_upgrade_connection` included the *entire* `SETTINGS` frame, instead of just the payload.
- Resolved issue where the `HTTP2-Settings` header value sent by a client for plaintext upgrade would be ignored by `initiate_upgrade_connection`, rather than have those settings applied appropriately.

1.11.15 2.4.3 (2017-01-27)

- Resolved issue where the `HTTP2-Settings` header value for plaintext upgrade that was emitted by `initiate_upgrade_connection` included the *entire* `SETTINGS` frame, instead of just the payload.
- Resolved issue where the `HTTP2-Settings` header value sent by a client for plaintext upgrade would be ignored by `initiate_upgrade_connection`, rather than have those settings applied appropriately.

1.11.16 2.3.4 (2017-01-27)

- Resolved issue where the `HTTP2-Settings` header value for plaintext upgrade that was emitted by `initiate_upgrade_connection` included the *entire* `SETTINGS` frame, instead of just the payload.
- Resolved issue where the `HTTP2-Settings` header value sent by a client for plaintext upgrade would be ignored by `initiate_upgrade_connection`, rather than have those settings applied appropriately.

1.11.17 2.5.1 (2016-12-17)

Bugfixes

- Remote peers are now allowed to send zero or any positive number as a value for `SETTINGS_MAX_HEADER_LIST_SIZE`, where previously sending zero would raise a `InvalidSettingsValueError`.

1.11.18 2.5.0 (2016-10-25)

API Changes (Backward-Compatible)

- Added a new `H2Configuration` object that allows rich configuration of a `H2Connection`. This object supersedes the prior keyword arguments to the `H2Connection` object, which are now deprecated and will be removed in 3.0.
- Added support for automated window management via the `acknowledge_received_data` method. See the documentation for more details.
- Added a `DenialOfServiceError` that is raised whenever a behaviour that looks like a DoS attempt is encountered: for example, an overly large decompressed header list. This is a subclass of `ProtocolError`.
- Added support for setting and managing `SETTINGS_MAX_HEADER_LIST_SIZE`. This setting is now defaulted to 64kB.

- Added h2.errors.ErrorCodes, an enum that is used to store all the HTTP/2 error codes. This allows us to use a better printed representation of the error code in most places that it is used.
- The error_code fields on ConnectionTerminated and StreamReset events have been updated to be instances of ErrorCodes whenever they correspond to a known error code. When they are an unknown error code, they are instead int. As ErrorCodes is a subclass of int, this is non-breaking.
- Deprecated the other fields in h2.errors. These will be removed in 3.0.0.

Bugfixes

- Correctly reject request header blocks with neither :authority nor Host headers, or header blocks which contain mismatched :authority and Host headers, per RFC 7540 Section 8.1.2.3.
- Correctly expect that responses to HEAD requests will have no body regardless of the value of the Content-Length header, and reject those that do.
- Correctly refuse to send header blocks that contain neither :authority nor Host headers, or header blocks which contain mismatched :authority and Host headers, per RFC 7540 Section 8.1.2.3.
- Hyper-h2 will now reject header field names and values that contain leading or trailing whitespace.
- Correctly strip leading/trailing whitespace from header field names and values.
- Correctly refuse to send header blocks with a TE header whose value is not trailers, per RFC 7540 Section 8.1.2.2.
- Correctly refuse to send header blocks with connection-specific headers, per RFC 7540 Section 8.1.2.2.
- Correctly refuse to send header blocks that contain duplicate pseudo-header fields, or with pseudo-header fields that appear after ordinary header fields, per RFC 7540 Section 8.1.2.1.

  This may cause passing a dictionary as the header block to send_headers to throw a ProtocolError, because dictionaries are unordered and so they may trip this check. Passing dictionaries here is deprecated, and callers should change to using a sequence of 2-tuples as their header blocks.
- Correctly reject trailers that contain HTTP/2 pseudo-header fields, per RFC 7540 Section 8.1.2.1.
- Correctly refuse to send trailers that contain HTTP/2 pseudo-header fields, per RFC 7540 Section 8.1.2.1.
- Correctly reject responses that do not contain the :status header field, per RFC 7540 Section 8.1.2.4.
- Correctly refuse to send responses that do not contain the :status header field, per RFC 7540 Section 8.1.2.4.
- Correctly update the maximum frame size when the user updates the value of that setting. Prior to this release, if the user updated the maximum frame size hyper-h2 would ignore the update, preventing the remote peer from using the higher frame sizes.

1.11.19 2.4.2 (2016-10-25)

Bugfixes

- Correctly update the maximum frame size when the user updates the value of that setting. Prior to this release, if the user updated the maximum frame size hyper-h2 would ignore the update, preventing the remote peer from using the higher frame sizes.

1.11.20 2.3.3 (2016-10-25)

Bugfixes

- Correctly update the maximum frame size when the user updates the value of that setting. Prior to this release, if the user updated the maximum frame size hyper-h2 would ignore the update, preventing the remote peer from using the higher frame sizes.

1.11.21 2.2.7 (2016-10-25)

Final 2.2.X release

Bugfixes

- Correctly update the maximum frame size when the user updates the value of that setting. Prior to this release, if the user updated the maximum frame size hyper-h2 would ignore the update, preventing the remote peer from using the higher frame sizes.

1.11.22 2.4.1 (2016-08-23)

Bugfixes

- Correctly expect that responses to HEAD requests will have no body regardless of the value of the Content-Length header, and reject those that do.

1.11.23 2.3.2 (2016-08-23)

Bugfixes

- Correctly expect that responses to HEAD requests will have no body regardless of the value of the Content-Length header, and reject those that do.

1.11.24 2.4.0 (2016-07-01)

API Changes (Backward-Compatible)

- Adds `additional_data` to `H2Connection.close_connection`, allowing the user to send additional debug data on the GOAWAY frame.
- Adds `last_stream_id` to `H2Connection.close_connection`, allowing the user to manually control what the reported last stream ID is.
- Add new method: `prioritize`.
- Add support for emitting stream priority information when sending headers frames using three new keyword arguments: `priority_weight`, `priority_depends_on`, and `priority_exclusive`.
- Add support for "related events": events that fire simultaneously on a single frame.

1.11.25 2.3.1 (2016-05-12)

Bugfixes

- Resolved AttributeError encountered when receiving more than one sequence of CONTINUATION frames on a given connection.

1.11.26 2.2.5 (2016-05-12)

Bugfixes

- Resolved AttributeError encountered when receiving more than one sequence of CONTINUATION frames on a given connection.

1.11.27 2.3.0 (2016-04-26)

API Changes (Backward-Compatible)

- Added a new flag to the H2Connection constructor: header_encoding, that controls what encoding is used (if any) to decode the headers from bytes to unicode. This defaults to UTF-8 for backward compatibility. To disable the decode and use bytes exclusively, set the field to False, None, or the empty string. This affects all headers, including those pushed by servers.
- Bumped the minimum version of HPACK allowed from 2.0 to 2.2.
- Added support for advertising RFC 7838 Alternative services.
- Allowed users to provide hpack.HeaderTuple and hpack.NeverIndexedHeaderTuple objects to all methods that send headers.
- Changed all events that carry headers to emit hpack.HeaderTuple and hpack.NeverIndexedHeaderTuple instead of plain tuples. This allows users to maintain header indexing state.
- Added support for plaintext upgrade with the initiate_upgrade_connection method.

Bugfixes

- Automatically ensure that all Authorization and Proxy-Authorization headers, as well as short Cookie headers, are prevented from being added to encoding contexts.

1.11.28 2.2.4 (2016-04-25)

Bugfixes

- Correctly forbid pseudo-headers that were not defined in RFC 7540.
- Ignore AltSvc frames, rather than exploding when receiving them.

1.11.29 2.1.5 (2016-04-25)

Final 2.1.X release

Bugfixes

- Correctly forbid pseudo-headers that were not defined in RFC 7540.
- Ignore AltSvc frames, rather than exploding when receiving them.

1.11.30 2.2.3 (2016-04-13)

Bugfixes

- Allowed the 4.X series of hyperframe releases as dependencies.

1.11.31 2.1.4 (2016-04-13)

Bugfixes

- Allowed the 4.X series of hyperframe releases as dependencies.

1.11.32 2.2.2 (2016-04-05)

Bugfixes

- Fixed issue where informational responses were erroneously not allowed to be sent in the `HALF_CLOSED_REMOTE` state.
- Fixed issue where informational responses were erroneously not allowed to be received in the `HALF_CLOSED_LOCAL` state.
- Fixed issue where we allowed information responses to be sent or received after final responses.

1.11.33 2.2.1 (2016-03-23)

Bugfixes

- Fixed issue where users using locales that did not default to UTF-8 were unable to install source distributions of the package.

1.11.34 2.2.0 (2016-03-23)

API Changes (Backward-Compatible)

- Added support for sending informational responses (responses with 1XX status) codes as part of the standard flow. HTTP/2 allows zero or more informational responses with no upper limit: hyper-h2 does too.
- Added support for receiving informational responses (responses with 1XX status) codes as part of the standard flow. HTTP/2 allows zero or more informational responses with no upper limit: hyper-h2 does too.
- Added a new event: `ReceivedInformationalResponse`. This response is fired when informational responses (those with 1XX status codes).
- Added an `additional_data` field to the `ConnectionTerminated` event that carries any additional data sent on the GOAWAY frame. May be `None` if no such data was sent.

- Added the `initial_values` optional argument to the `Settings` object.

Bugfixes

- Correctly reject all of the connection-specific headers mentioned in RFC 7540 § 8.1.2.2, not just the `Connection:` header.
- Defaulted the value of `SETTINGS_MAX_CONCURRENT_STREAMS` to 100, unless explicitly overridden. This is a safe defensive initial value for this setting.

1.11.35 2.1.3 (2016-03-16)

Deprecations

- Passing dictionaries to `send_headers` as the header block is deprecated, and will be removed in 3.0.

1.11.36 2.1.2 (2016-02-17)

Bugfixes

- Reject attempts to push streams on streams that were themselves pushed: streams can only be pushed on streams that were initiated by the client.
- Correctly allow CONTINUATION frames to extend the header block started by a PUSH_PROMISE frame.
- Changed our handling of frames received on streams that were reset by the user.

  Previously these would, at best, cause ProtocolErrors to be raised and the connection to be torn down (rather defeating the point of resetting streams at all) and, at worst, would cause subtle inconsistencies in state between hyper-h2 and the remote peer that could lead to header block decoding errors or flow control blockages.

  Now when the user resets a stream all further frames received on that stream are ignored except where they affect some form of connection-level state, where they have their effect and are then ignored.
- Fixed a bug whereby receiving a PUSH_PROMISE frame on a stream that was closed would cause a RST_STREAM frame to be emitted on the closed-stream, but not the newly-pushed one. Now this causes a `ProtocolError`.

1.11.37 2.1.1 (2016-02-05)

Bugfixes

- Added debug representations for all events.
- Fixed problems with setup.py that caused trouble on older setuptools/pip installs.

1.11.38 2.1.0 (2016-02-02)

API Changes (Backward-Compatible)

- Added new field to `DataReceived`: `flow_controlled_length`. This is the length of the frame including padded data, allowing users to correctly track changes to the flow control window.

- Defined new UnsupportedFrameError, thrown when frames that are known to hyperframe but not supported by hyper-h2 are received. For backward-compatibility reasons, this is a ProtocolError *and* a KeyError.

Bugfixes

- Hyper-h2 now correctly accounts for padding when maintaining flow control windows.
- Resolved a bug where hyper-h2 would mistakenly apply SETTINGS_INITIAL_WINDOW_SIZE to the connection flow control window in addition to the stream-level flow control windows.
- Invalid Content-Length headers now throw ProtocolError exceptions and correctly tear the connection down, instead of leaving the connection in an indeterminate state.
- Invalid header blocks now throw ProtocolError, rather than a grab bag of possible other exceptions.

1.11.39 2.0.0 (2016-01-25)

API Changes (Breaking)

- Attempts to open streams with invalid stream IDs, either by the remote peer or by the user, are now rejected as a ProtocolError. Previously these were allowed, and would cause remote peers to error.
- Receiving frames that have invalid padding now causes the connection to be terminated with a ProtocolError being raised. Previously these passed undetected.
- Settings values set by both the user and the remote peer are now validated when they're set. If they're invalid, a new InvalidSettingsValueError is raised and, if set by the remote peer, a connection error is signaled. Previously, it was possible to set invalid values. These would either be caught when building frames, or would be allowed to stand.
- Settings changes no longer require user action to be acknowledged: hyper-h2 acknowledges them automatically. This moves the location where some exceptions may be thrown, and also causes the acknowledge_settings method to be removed from the public API.
- Removed a number of methods on the H2Connection object from the public, semantically versioned API, by renaming them to have leading underscores. Specifically, removed:
  - get_stream_by_id
  - get_or_create_stream
  - begin_new_stream
  - receive_frame
  - acknowledge_settings
- Added full support for receiving CONTINUATION frames, including policing logic about when and how they are received. Previously, receiving CONTINUATION frames was not supported and would throw exceptions.
- All public API functions on H2Connection except for receive_data no longer return lists of events, because these lists were always empty. Events are now only raised by receive_data.
- Calls to increment_flow_control_window with out of range values now raise ValueError exceptions. Previously they would be allowed, or would cause errors when serializing frames.

API Changes (Backward-Compatible)

- Added PriorityUpdated event for signaling priority changes.
- Added get_next_available_stream_id function.
- Receiving DATA frames on streams not in the OPEN or HALF_CLOSED_LOCAL states now causes a stream reset, rather than a connection reset. The error is now also classified as a StreamClosedError, rather than a more generic ProtocolError.
- Receiving HEADERS or PUSH_PROMISE frames in the HALF_CLOSED_REMOTE state now causes a stream reset, rather than a connection reset.
- Receiving frames that violate the max frame size now causes connection errors with error code FRAME_SIZE_ERROR, not a generic PROTOCOL_ERROR. This condition now also raises a FrameTooLargeError, a new subclass of ProtocolError.
- Made NoSuchStreamError a subclass of ProtocolError.
- The StreamReset event is now also fired whenever a protocol error from the remote peer forces a stream to close early. This is only fired once.
- The StreamReset event now carries a flag, remote_reset, that is set to True in all cases where StreamReset would previously have fired (e.g. when the remote peer sent a RST_STREAM), and is set to False when it fires because the remote peer made a protocol error.
- Hyper-h2 now rejects attempts by peers to increment a flow control window by zero bytes.
- Hyper-h2 now rejects peers sending header blocks that are ill-formed for a number of reasons as set out in RFC 7540 Section 8.1.2.
- Attempting to send non-PRIORITY frames on closed streams now raises StreamClosedError.
- Remote peers attempting to increase the flow control window beyond 2**31 - 1, either by window increment or by settings frame, are now rejected as ProtocolError.
- Local attempts to increase the flow control window beyond 2**31 - 1 by window increment are now rejected as ProtocolError.
- The bytes that represent individual settings are now available in h2.settings, instead of needing users to import them from hyperframe.

Bugfixes

- RFC 7540 requires that a separate minimum stream ID be used for inbound and outbound streams. Hyper-h2 now obeys this requirement.
- Hyper-h2 now does a better job of reporting the last stream ID it has partially handled when terminating connections.
- Fixed an error in the arguments of StreamIDTooLowError.
- Prevent ValueError leaking from Hyperframe.
- Prevent struct.error and InvalidFrameError leaking from Hyperframe.

1.11.40 1.1.1 (2015-11-17)

Bugfixes

- Forcibly lowercase all header names to improve compatibility with implementations that demand lower-case header names.

1.11.41 1.1.0 (2015-10-28)

API Changes (Backward-Compatible)

- Added a new `ConnectionTerminated` event, which fires when GOAWAY frames are received.
- Added a subclass of `NoSuchStreamError`, called `StreamClosedError`, that fires when actions are taken on a stream that is closed and has had its state flushed from the system.
- Added `StreamIDTooLowError`, raised when the user or the remote peer attempts to create a stream with an ID lower than one previously used in the dialog. Inherits from `ValueError` for backward-compatibility reasons.

Bugfixes

- Do not throw `ProtocolError` when attempting to send multiple GOAWAY frames on one connection.
- We no longer forcefully change the decoder table size when settings changes are ACKed, instead waiting for remote acknowledgement of the change.
- Improve the performance of checking whether a stream is open.
- We now attempt to lazily garbage collect closed streams, to avoid having the state hang around indefinitely, leaking memory.
- Avoid further per-stream allocations, leading to substantial performance improvements when many short-lived streams are used.

1.11.42 1.0.0 (2015-10-15)

- First production release!

Python Module Index

Index