Scrapinghub (Scrapy) : HTTP/2 Support

About Me

Contact Information

Name	Aditya Kumar
Country	India
University	**National Institute of Technology, Tiruchirappalli**
Course	Bachelor of Technology in Computer Science and Engineering
Expected Grad. Date	June 2021
Timezone	IST (GMT +5:30)
Email	**k.aditya00@gmail.com**
Resume	**Link to latest resume**
Github	**github.com/adityaa30**

Code Contributions

Issues

- 👉 #4444 - Update redirect link from Python 2 docs to Python 3 docs

Pull Requests

- 👉 #4377 - [wip] option to output data in scrapy parse
- 👉 #4390 - [docs] async/deferred signal handlers
- 👉 #4406 - [fix] wrong reactor installed error
- 👉 #4445 - [docs] update redirect links to python3
- 👉 #4448 - [fix] zope interface 5.0.0 unsupported

Project Information

Sub-org name

`Scrapinghub - Scrapy`

Proposal

This project aims to implement HTTP/2 support adding an HTTP handler that can gracefully upgrade to HTTP/2 where possible, and take advantage of the compression and efficient gains.

HTTP/2 will make Scrapy faster, simpler, and more robust — a rare combination — by allowing us to undo many of the HTTP/1.1 workarounds previously done within Scrapy and address these concerns within the transport layer itself.

Disadvantages of using HTTP/1.x based client:

- 👉 Need of multiple connections to achieve concurrency and reduce latency
- 👉 No compression for request and response headers adding up unnecessary network traffic
- 👉 No effective resource prioritization, resulting in poor use of the underlying TCP connection

hyper-h2 contains a pure-python implementation of a HTTP/2 protocol stack with a simple yet powerful API. We can use this as a building block for the complete HTTP/2 Client implementation.

Benefits earned by implementing HTTP/2 client will be:

- 👉 HTTP/2 introduces HPACK (**h2** uses **hpack** internally) introducing compression of headers fields in request/response automatically enabling a more efficient use of network resources and a reduced perception of latency.
- 👉 HTTP/2 being binary (rather than text as in HTTP/1.1) enables more efficient processing of messages (*via binary message framing*)
- 👉 Multiple concurrent exchanges on the same connection (*multiplexing*)
- 👉 Fewer TCP connections are used in comparison to HTTP/1.x i.e less competition with other flows and longer-lived connections, which in turn leads to a better utilization of available network capacity

- 👉 HTTP/2 have the same application semantics of HTTP and core concepts (eg. HTTP methods, status codes, URIs, header fields) allowing us to use Scrapy's inbuilt `Request`, `Response`, etc without altering the current implementation
- 👉 Eliminate unnecessary latency and HTTP/1.x workarounds, such as concatenated files, image sprites and domain sharding

The two key upgrades that we will achieve with HTTP/2 will be

- 👉 **Single connection per resource** - HTTP/2 uses one connection per resource, instead of one connection per file request. This means much less need for time-consuming connection setup, which is especially beneficial with TLS, because TLS connections are particularly time-consuming to create.
- 👉 **Faster TLS performance** - HTTP/2 only needs one expensive TLS handshake, and multiplexing gets the most out of the single connection. As HTTP/2 will compress header data, and avoiding HTTP/1.1 optimizations such as file concatenation makes caching work more efficiently

Implementation

- 👉 As Hyper-h2 encodes the actions of the remote peer in the form of events. When we receive data from the remote peer and pass it into `H2Connection` object a list of objects will be returned, each one representing a single event that has occurred. Each event refers to a single action the remote peer has taken. As we will be tackling all the events in general, an interface will be introduced listing all the important methods:

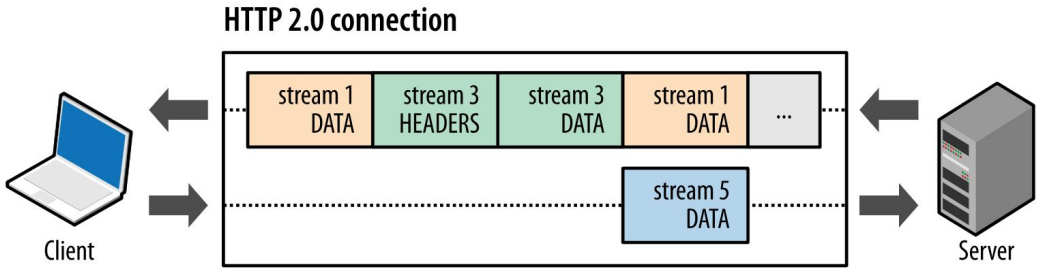
```
from zope.interface import Interface

class IH2EventsHandler(Interface):
    📌 response_received(event: ResponseReceived)
    📌 trailers_received(event: TrailersReceived)
    📌 data_received(event: DataReceived)
    📌 window_updated(event: WindowUpdated)
    📌 stream_ended(event: StreamEnded)
    📌 stream_reset(event: StreamReset)
    📌 connection_terminated(event: ConnectionTerminated)
```

- 👉 Now, `IH2EventsHandler` will be implemented by `H2ClientProtocol`.

  I am planning to have the implementation of this protocol to be inspired by Twisted `HTTP11ClientProtocol`.

  We can benefit from the **binary framing layer** in HTTP/2 protocol using only **one long-lived TCP connection per server** between our client and server. As **binary framing**

layer enables full request and response multiplexing, by allowing the client and server to break down an HTTP message into independent frames, interleave them and reassemble them on the other end (refer below image)



The HTTP/2 multiplexing

Image Source: Introduction to HTTP/2 | Web Fundamentals

We will implement above above behaviour -

- Terminology
  - **Stream**: A bidirectional flow of bytes within an established connection, which may carry one or more messages.
  - **Message**: A complete sequence of frames that map to a logical request or response message. It is a logical HTTP message, such as a request, or response, which consists of one or more frames.
  - **Frame**: The smallest unit of communication in HTTP/2, each containing a frame header, which at a minimum identifies the stream to which the frame belongs.
- Interleave multiple requests and incoming responses in parallel without blocking on any one using a single long-lived connection that can carry any number of bidirectional streams
- Each stream has a unique identifier which will be used to carry bidirectional message
- A `dict` will be maintained representing a pool of streams where
  - Key {`int`} -- Bidirectional stream unique identifier (`stream_id`)
  - Value {`Response`} -- Data delivered by a frame
- With each `request` method call, a new entry in above defined dict will be added and updated at each time some data/response is received
- When the the response for a request is received completely via a stream, the final response received will be passed using the `Deferred` returned by `request` method

```
from twisted.internet.protocol import Protocol
from zope.interface import implementer
```

```
@implementer(IH2EventHandler)
class H2ClientProtocol(Protocol):
```

- 📌 `__init__ (callback)`
  - 📍 Parameters
    - → callback {`callable`} -- Callback to inform `H2ConnectionPool` about the status of handshake. Contains `connection: H2ClientProtocol` (`self`) instance as an argument which is either closed gracefully or kept back by the `H2ConnectionPool` similar to the implementation of pool in `HTTPConnectionPool`
  - 📍 Instantiated by `H2ClientFactory`
  - 📍 Creates an instance of `H2Connection` as `self.conn`
- 📌 `request (_request: Request)`
  - 📍 Parameters
    - → _request {`scrapy.http.Request`} -- object containing all the parameters of the network request to be issued
  - 📍 Return a `Deferred` that will fire with a `scrapy.http.Response` instance or an error
  - 📍 Issue `_request` over `self.transport`
- 📌 `connectionMade ()`
  - 📍 Method defined by `BaseProtocol` considered as the initializer of the protocol, because it is called when the connection to the server is established successfully
  - 📍 Calls `self.conn.initiate_connection` (if connection is not initiated already) to ensure that the appropriate preamble data is placed in the data buffer
- 📌 `dataReceived (data)`
  - 📍 Method defined by class `Protocol`
  - 📍 Get the events received to `self.conn` and call `handle_event`
- 📌 `handle_event (events)`
  - 📍 Parameter
    - → event {`list`} -- List of event object returned by `H2Connection`
  - 📍 This method will act as a bridge between the `events` received from the HTTP/2 data and the methods defined in `IH2EventsHandler`
- 📌 <methods of `IH2EventsHandler`>
- 📌 `connectionLost (reason)`
  - 📍 Method defined by class `Protocol`
  - 📍 Called by Twisted when the connection is gone i.e the underlying `self.transport` went away.

👉 We will use **Hyper-h2**'s `H2Connection` to represent the state of a single HTTP/2 connection. As per Hyper-h2 docs `H2Connection` *object will be the first thing you create*

and the object that does most of the heavy lifting.

Now, we will introduce a class `H2ConnectionPool` having its structure similar to Twisted `HTTPConnectionPool`. The difference here will be that `get_connection` method will return a `Deferred` that will fire with our `H2ClientProtocol` which will be used to send a **HTTP/2** request. `H2ClientProtocol` will be provided by `H2ClientFactory` mentioned below.

`H2ConnectionPool` will:

- Maintain a pool of persistent `H2Connection`'s
- The pool will be implemented using python dictionary, where
  - Key -- argument provided in `get_connection`. The key is created by the `H2Agent` to uniquely identify a connection
  - Value -- `H2ClientProtocol` instance
- Limits the number of persistent connections
- Connections will be stored using keys, which should be chosen such that any connections stored under a given key can be used interchangeably
- Failed requests done using previously cached connections will be retried once if they use an idempotent method (e.g. GET), in case the HTTP server timed them out.

`class H2ConnectionPool:`

- `__init__ (reactor, conn_limit=2)`
  - Parameters
    - → reactor -- A reactor to add event to close connection
    - → conn_limit {`int`} -- The maximum number of cached persistent connections for a (host:port) destination (2 by default)
- `get_connection (key, endpoint)`
  - Parameters
    - → key {`tuple`} -- unique key identifying connections that can be used interchangeably
    - → endpoint -- endpoint that can be used to open a new connection if no cached connection is available
  - Supplies a connection (either new or retrieved from cached connections) which will be used for multiple HTTP/2 requests based on the key. The connection will remain inside the pool. In case, `conn_limit` is reached and there is a requirement of a new connection, one of the connections will be terminated and replaced by a new connection to fulfill the requirement.
  - LRU algorithm will be used for connection termination in case of a tie mentioned above
  - Returns a `Deferred` which will fire with `H2ClientProtocol` which will be used to send a single HTTP/2 request

→ `return endpoint.connect(`*<Instance of* `H2ClientFactory`*>*`)`

📌 `close_cached_connections ()`

📍 Functionality similar to Twisted `HTTPConnectionPool.closeCachedConnections`

👉 As mentioned above, class `H2ClientFactory` will be required by `H2ConnectionPool` to provide the `endpoint` with a protocol instance while connecting to the server.

`class H2ClientFactory(protocol.Factory):`

📌 `__init__ (callback, metadata)`

📍 Parameters

→ callback {`callable`} -- Callback to know the status of the request is handled completely. Contains `connection: H2ClientProtocol` instance as an parameter which is either closed gracefully or kept back in the `H2ConnectionPool`

→ metadata {`str`} -- Metadata about the low-level connection details, used to make the repr more useful

📌 `buildProtocol (addr)`

📍 Simply initializes an instance of `H2ClientProtocol` and return it

👉 Since Twisted `Agent` supports only upto HTTP/1.1, we need to create our own custom class `H2Agent`. `H2Agent` will be responsible for -

📌 Resolving the hostname into an IP address and connecting it on the respective port

📌 Verifies that the `url` provided by the `request` is valid

📌 Get `uri` from `request.url` and issue a request to the server indicated by the parsed `uri` with a connection from the `pool: H2ConnectionPool` and an `endpoint` using `self.endpoint_factory`

📌 Sending multiple requests concurrently by multiplexing the request in different streams on the same **long-lived TCP connection** established between our client and server

📌 Generates a key using the `uri` of the location of the server to request

`class H2Agent:`

📌 `__init__ (_reactor, pool, context_factory, connect_timeout=None)`

📍 Parameters

→ _reactor -- A reactor for the `H2Agent` to place outgoing connections (cannot be None)

→ pool {`H2ConnectionPool`}

→ context_factory {`IPolicyForHTTPS`} -- A factory for TLS contexts, to control the verification parameters of OpenSSL

→ connect_timeout {`int`} -- The amount of time that `H2Agent` will wait for the peer to accept a connection

📍 Here we will initialize a class variable `self.endpoint_factory` using class implementing `IAgentEndpointFactory`

📌 `request (_request: Request)`

📍 Parameters

→ _request {`scrapy.http.Request`} -- object containing all the parameters of the network request to be issued

📍 Calls the `request` method of `self.pool` passing `_request`

📍 Returns a `Deferred` that fires with `scrapy.http.Response` when the response has been received (regardless of the response status code) or with a `Failure` if there is any problem which prevents that response from being received (including problems that prevent the request from being sent)

📌 `get_key (uri: URI)`

📍 Parameter

→ uri {`bytes`} -- The location of the server to request. This should be an absolute URI.

📍 Return a `tuple(uri.scheme, uri.host, uri.port)` which is used by `H2ConnectionPool` to uniquely identify a connection

👉 A class `H2DownloadHandler` will be introduced *similar to* `HTTP11DownloadHandler`.

Here is a description of how `H2DownloadHandler` class will look like:

```
class H2DownloadHandler:
```

📌 `__init__ (settings, crawler=None)`

📍 Parameters (Similar to `HTTP11DownloadHandler`)

→ settings - scrapy.settings.Settings object

→ crawler - Optional

📍 Similar to `HTTP11DownloadHandler`, an instance of `H2ConnectionPool` will be created

📌 `from_crawler (crawler) # classmethod`

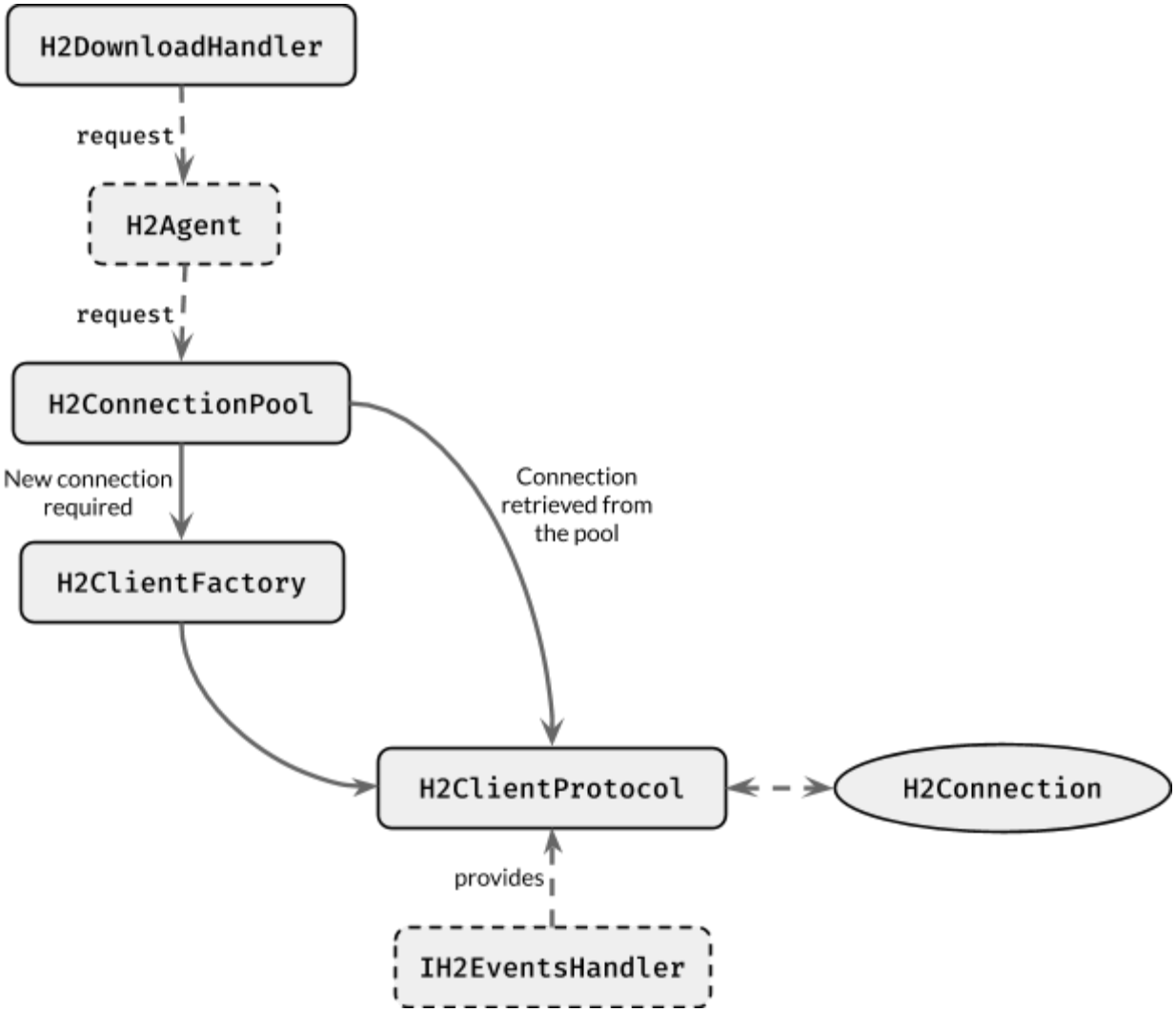
📍 Create `H2DownloadHandler` object using Scrapy `Crawler` instance

📌 `download_request (request, spider)`

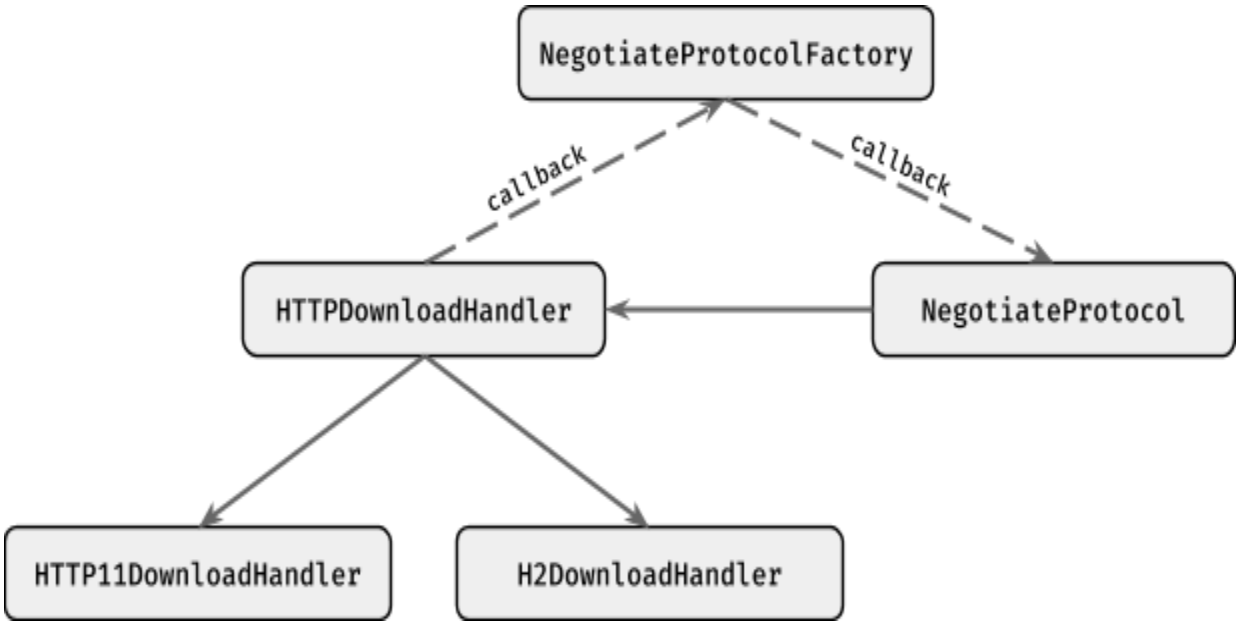
📍 Parameters description & implementation similar to `HTTP11DownloadHandler.download_request(request, spider)` in Scrapy

📍 However instead of instantiating `ScrapyAgent`, `H2Agent` will be instantiated passing the `request` argument directly!

Above implementation can be summarised by the graph below



Integration of H2 in Scrapy



Integration of H2 in Scrapy will primarily focus on `H2DownloadHandler` and `HTTP11DownloadHandler`. As both of these API have their own respective protocols, we require an architecture which can

- 👉 Decide a suitable handler for a particular request automatically
- 👉 Clean and simple to understand
- 👉 Minimal changes to the existing codebase required

In the image provided, the architecture is proposed which introduces following three new classes

- 👉 `class HTTPDownloadHandler:`
  - 📌 `__init__ (settings, crawler=None)`
    - 📍 Parameters
      - → settings {`Settings`}
      - → crawler {`Crawler`}
    - 📍 Instantiates both download handlers
  - 📌 `from_crawler (crawler) # classmethod`
    - 📍 Create `HTTPDownloadHandler` object using Scrapy `Crawler` instance
  - 📌 `update_protocol (request, protocol)`
    - 📍 This function acts as a callback and when called with the information about the negotiated callback, it will update the respective HTTP handler by adding the `protocol` instance in the connection pool
    - 📍 After this the `request` is sent using the respective handler

- 📌 `check_conn_exist (request: Request)`
  - 📍 Using the pools in both of the download handlers, it checks if a persistent connection to the request `uri` already exists. If it does exist, then it returns that pool's instance or `None`.
- 📌 `download_request (request, spider)`
  - 📍 Calls `self.check_conn_exist` to verify if a new connection is required or a connection can directly be used from the pools
  - 📍 Based on the above result
    - → If a connection already persists then we can pass the parameters to the respective handler and handle the request
    - → Else initiate a connection for the respective resource by passing the setting `ACCEPTED_PROTOCOLS` (mentioned below). This function wraps the `update_protocol` method in a callback such that negotiated protocol can be notified

👉 `class NegotiateProtocolFactory(protocol.Factory):`

- 📌 `__init__ (callback, metadata)`
  - 📍 Parameters
    - → callback {`callable`} -- Callback to inform `HTTPDownloadHandler` about the protocol which was negotiated via Twisted. It is passed down to `NegotiateProtocol` as shown in the above image.
    - → metadata {`str`} -- Metadata about the low-level connection details, used to make the repr more useful
- 📌 `buildProtocol ()`
  - 📍 Simply initializes an instance of `NegotiateProtocol` and return it

👉 `class NegotiateProtocol(protocol.Protocol):`

- 📌 `__init__(callback)`
  - 📍 Parameters
    - → callback {`callable`} -- Callback to inform `HTTPDownloadHandler` about the protocol which was negotiated via Twisted
- 📌 `connectionMade ()`
  - 📍 Based on the protocol negotiated during the handshake (`self.transport.negotiatedProtocol`) we can use the callback now to pass this information to `HTTPDownloadHandler`
  - 📍 Updated the underlying protocol (`self.transport.wrappedProtocol`) based on the negotiated protocol

Setting Attributes

A new `Settings` attribute shall be introduced

- 👉 `ACCEPTED_PROTOCOLS`
  - 📌 Will be passed as a parameter to the `optionsForClientTLS()` function by Scrapy's `BrowserLikeContextFactory`
  - 📌 Type -- `list<bytes>` or `bytes` (Can be `None`)
  - 📌 Default value can be -- `[b'h2', b'http/1.1', b'http/1.0']`
  - 📌 List of protocols that the peer is willing to speak after the TLS negotiation has completed, advertised over both ALPN and NPN by default in Twisted.
  - 📌 If this setting is specified, and no overlap can be found with the other peer, the connection will fail to be established (default behaviour in Twisted). If the remote peer does not offer NPN or ALPN, the connection will be established with HTTP/1.x protocol if possible.
  - 📌 Protocols earlier in the list will be preferred over those later in the list
  - 📌 This setting will be effective only when `HTTPDownloadHandler` is enabled as it will have the flexibility to support both HTTP/1.x and HTTP/2 protocols simultaneously.
  - 📌 However, if the setting `DOWNLOAD_HANDLERS` is modified then the above scheme will not be effective

Regarding other setting attributes such as `DOWNLOAD_HANDLERS`, `DOWNLOADER_HTTPCLIENTFACTORY`, `DOWNLOADER_CLIENTCONTEXTFACTORY`, `DOWNLOADER_CLIENT_TLS_*` will not require any major change. However, A small change may be introduced in the default classes in Scrapy to support `ACCEPTED_PROTOCOLS`

Work Timeline

Note: All dates below are for year 2020

Community Bonding Period : 4th May - 31st May

4th May - 6th May (~3 days)

- 👉 Decide on schedule and mode of communication for weekly and emergency meetings
- 👉 Setup other logistic i.e GSoC blog

7th May - 20th May (~2 weeks)

- 👉 Discuss with mentors about any changes or improvement to be made in proposed architecture and draft a final design document

- 👉 Finalise proposed architecture, deadlines and milestones with mentors
- 👉 Increase my proficiency in Scrapy and Twisted
- 👉 Increase familiarity with community practises and processes
- 👉 Familiarize myself more with the Scrapy codebase

21st May - 27th May (~1 week)

- 👉 Setup development environment
- 👉 Start a discussion for the implementation of `IH2EventsHandler` in `H2ClientProtocol`

Coding Phase

Week 1 : 28th May - 7th June

- 👉 Discuss with the mentors about the methods defined in `IH2EventsHandler`
- 👉 This discussion will be solely to understand if there is anything left out of HTTP/2 that can be added
- 👉 Finalize the methods & their expected behaviour for `IH2EventsHandler` interface
- 👉 Add the code for `IH2EventsHandler` interface declaration with detailed docstring & documentation done for each method

Week 2 & 3: 8th June - 21st June

- 👉 Start writing code for `H2ClientProtocol`. Since this will be a relatively more important part of this project, more time is scheduled for it.
- 👉 Add docstring for methods defined in `H2ClientProtocol` as the methods are implemented
- 👉 Discuss with mentors regarding the implementation done regularly (at least once in every 2-3 days)
- 👉 Look for further optimizations
- 👉 Add unit tests for `H2ClientProtocol`

Week 4 : 22nd June - 28th June

- 👉 Add documentation for `H2ClientProtocol` and finish writing unit tests
- 👉 This week is kept as a buffer week to complete any leftover (if any) work

Week 5 : 29th June - 5th July

- 👉 Start writing code for `H2ConnectionPool` and `H2ClientFactory`
- 👉 Discuss about the possible ***connection termination*** algorithms in `H2ConnectionPool` with the mentors

- 👉 Implement the algorithm finalized in the discussion
- 👉 Add docstring for methods and functions implemented

Week 6 : 6th July - 12th July

- 👉 Add unit tests for `H2ConnectionPool` and `H2ClientFactory`
- 👉 Add documentation for both classes

Week 7 : 13th July - 19th July

- 👉 Start writing code for `H2Agent`
- 👉 Discuss with mentors regarding the implementation of `TunnelingTCP4ClientEndpoint`, `TunnelingAgent`, `ScrapyProxyAgent` and `ScrapyAgent` as they are currently implented in Scrapy for HTTP/1.x subclassing Twisted `Agent`
- 👉 The discussion will finalize the implementation of above agents for HTTP/2
- 👉 Add docstring for methods and functions implemented

Week 8 : 20th July - 26th July

- 👉 Add unit tests for `H2Agent`
- 👉 Add documentation for `H2Agent`

Week 9 : 27th July - 2nd August

- 👉 Integrate `H2Agent`, `H2ConnectionPool` and `H2Protocol` together
- 👉 Write integration tests to verify the combined functionality after integration

Week 10 : 3rd August - 9th August

- 👉 Start writing code for `H2DownloadHandler`
- 👉 Discuss with mentors on integration of `HTTP/2` in Scrapy
- 👉 The discussion will finalize the integration process of `HTTP/2` in Scrapy
- 👉 Add docstring for methods and functions implemented
- 👉 Add unit tests for `H2DownloadHandler`

Week 11 : 10th August - 17th August

- 👉 Add documentation for `H2DownloadHandler`
- 👉 Complete integration of `H2DownloadHandler` in Scrapy
- 👉 Start writing code for `HTTPDownloadHandler` , `NegotiateProtocolFactory`, `NegotiateProtocol`
- 👉 Integrate all the components in Scrapy

- 👉 Add unit tests for `HTTPDownloadHandler`, `NegotiateProtocolFactory`, `NegotiateProtocol`

Week 12 : 18th August - 23rd August

- 👉 Write integration tests to verify the combined functionality after integration
- 👉 Perform black box testing and beta testing for the entire project

Week 13 : 24th August - 31st August (Code Submission & Final Evaluations)

- 👉 Complete any left over work

Future Work

HTTP/2 implementation doesn't stop with GSoC'20. Once the initial base is established, I want to keep working along further development and maintenance in my spare time. Some features I hope to work on after GSoC are as follows

- 👉 **Connection termination algorithm**
  - 📌 Algorithm to decide which one of the connections will be terminated in a connection pool and replaced by a new connection to fulfill the requirement when the connection limit is reached.
  - 📌 Switch to a more optimized and better algorithm compared to LRU
  - 📌 Add support for multiple configurable algorithms
- 👉 **Server Push**
  - 📌 Take advantage of server push at the client side efficiently
  - 📌 `PushedStreamReceived` event can be used to implement server push

More About Me

Hi, my name is Aditya Kumar and I am a student pursuing Bachelor of Technology in Computer Science and Engineering, currently in my junior year, from National Institute of Technology, Tiruchirappalli. I am an enthusiastic full-stack developer at my college's premier coding club, `Delta Force`.

I like to work, travel, play games (both indoor & outdoor) and especially listen to music. I've developed a crazy interest in 'coding' in general. I enjoy competitive programming and software development, attending hackathons and working on projects that make life a little easier and hacking on random things . I have been involved in software development, computer vision & artificial Intelligence for the past 2 years and currently working on improving my skills, finding new ways of thinking & problem solving to stay on top of development.

Other Commitments

- 👉 I have only applied for Scrapy in GSoC 2020
- 👉 I will be having semester examination, dates aren't fixed due to the pandemic. The examination period will be 15 days. I will lessen the workload during my exams by 3/4 of a regular.
- 👉 I will inform my mentors about any change in schedule in a timely manner

Thanks a lot 🙂