wsproto Documentation

Benno Rice

May 26, 2019

Contents

wsproto is a WebSocket protocol stack written to be as flexible as possible. To that end it is written in pure Python and performs no I/O of its own. Instead it relies on the user to provide a bridge between it and whichever I/O mechanism is in use, allowing it to be used in single-threaded, multi-threaded or event-driven code.

The goal for wsproto is 100% compliance with RFC 6455. Additionally a mechanism is provided to add extensions allowing the implementation of extra functionally such as per-message compression as specified in RFC 7692.

For usage examples, see *Getting Started* or see the examples provided.

Contents:

CHAPTER 1

Installation

wsproto is a pure Python project. To install it you can use pip like so:

```
$ pip install wsproto
```

Alternatively you can get either a release tarball or a development branch from our GitHub repository and run:

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

CHAPTER 2

Getting Started

This document explains how to get started using wsproto to connect to WebSocket servers as well as how to write your own.

We assume some level of familiarity with writing Python and networking code. If you're not familiar with these we highly recommend you read up on these first. It may also be helpful to study Sans-I/O, which describes the ideas behind writing a network protocol library that doesn't do any network I/O.

2.1 Connections

The main class you'll be working with is the `WSConnection` object. This object represents a connection to a WebSocket client or server and contains all the state needed to communicate with the entity at the other end. Whether you're connecting to a server or receiving a connection from a client, this is the object you'll use.

wsproto provides two layers of abstractions. You need to write code that interfaces with both of these layers. The following diagram illustrates how your code is like a sandwich around *wsproto*.

Application
<APPLICATION GLUE>
wsproto
<NETWORK GLUE>
Network Layer

wsproto does not do perform any network I/O, so `<NETWORK GLUE>` represents the code you need to write to glue *wsproto* to the actual network layer, i.e. code that can send and receive data over the network. The `WSConnection` class provides two methods for this purpose. When data has been received on a network socket, you feed this data into *wsproto* by calling `receive_data`. When *wsproto* sends events the `send` will return the bytes that need to be sent over the network. Your code is responsible for actually sending that data over the network.

Note: If the connection drops, a standard Python `socket.recv()` will return zero. You should call `receive_data(None)` to update the internal *wsproto* state to indicate that the connection has been closed.

Internally, *wsproto* process the raw network data you feed into it and turns it into higher level representations of WebSocket events. In <APPLICATION GLUE>, you need to write code to process these events. The *WSConnection* class contains a generator method `events` that yields WebSocket events. To send a message, you call the `send` method.

2.2 Connecting to a WebSocket server

Begin by instantiating a connection object in the client mode and then create a *Request* instance to send. The Request must specify `host` and `target` arguments. If the WebSocket server is located at `http://example.com/foo`, then you would instantiate the connection as follows:

```
ws = WSConnection(ConnectionType.CLIENT)
ws.send(Request(host="example.com", target='foo'))
```

Now you need to provide the network glue. For the sake of example, we will use standard Python sockets here, but *wsproto* can be integrated with any network layer:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("example.com", 80))
```

To read from the network:

```
data = sock.recv(4096)
ws.receive_data(data)
```

You also need to send data returned by the send method:

```
data = ws.send(Message(data=b"Hello"))
sock.send(data)
```

A standard Python socket will block on the call to `sock.recv()`, so you will probably need to use a non-blocking socket or some form of concurrency like threading, greenlets, asyncio, etc.

You also need to provide the application glue. To send a WebSocket message:

```
ws.send(Message(data="Hello world!"))
```

And to receive WebSocket events:

```
for event in ws.events():
    if isinstance(event, AcceptConnection):
        print('Connection established')
    elif isinstance(event, RejectConnection):
        print('Connection rejected')
    elif isinstance(event, CloseConnection):
        print('Connection closed: code={} reason={}'.format(
            event.code, event.reason
        ))
        sock.send(ws.send(event.response()))
    elif isinstance(event, Ping):
        print('Received Ping frame with payload {}'.format(event.payload))
        sock.send(ws.send(event.response()))
    elif isinstance(event, TextMessage):
        print('Received TEXT data: {}'.format(event.data))
        if event.message_finished:
```

(continues on next page)

(continued from previous page)

```
            print('Message finished.')
    elif isinstance(event, BytesMessage):
        print('Received BINARY data: {}'.format(event.data))
        if event.message_finished:
            print('BINARY Message finished.')
    else:
        print('Unknown event: {!r}'.format(event))
```

The method `events()` returns a generator which will yield events for all of the data currently in the *wsproto* internal buffer and then exit. Therefore, you should iterate over this generator after receiving new network data.

For a more complete example, see synchronous_client.py.

2.3 WebSocket Servers

A WebSocket server is similar to a client except that it uses a different constant:

```
ws = WSConnection(ConnectionType.SERVER)
```

A server also needs to explicitly send an *`AcceptConnection`* after it receives a `Request` event:

```
for event in ws.events():
    if isinstance(event, Request):
        print('Accepting connection request')
        sock.send(ws.send(AcceptConnection()))
    elif isinstance(event, CloseConnection):
        print('Connection closed: code={} reason={}'.format(
            event.code, event.reason
        ))
        sock.send(ws.send(event.response()))
    elif isinstance(event, Ping):
        print('Received Ping frame with payload {}'.format(event.payload))
        sock.send(ws.send(event.response()))
    elif isinstance(event, TextMessage):
        print('Received TEXT data: {}'.format(event.data))
        if event.message_finished:
            print('TEXT Message finished.')
    elif isinstance(event, BinaryMessage):
        print('Received BINARY data: {}'.format(event.data))
        if event.message_finished:
            print('BINARY Message finished.')
    else:
        print('Unknown event: {!r}'.format(event))
```

Alternatively a server can explicitly reject the connection by sending *`RejectConnection`* after receiving a `Request` event.

For a more complete example, see synchronous_server.py.

2.4 Protocol Errors

Protocol errors relating to either incorrect data or incorrect state changes are raised when the connection receives data or when events are sent. A *`LocalProtocolError`* is raised if the local actions are in error whereas a

RemoteProtocolError is raised if the remote actions are in error.

2.5 Closing

WebSockets are closed with a handshake that requires each endpoint to send one frame and receive one frame. Sending a *CloseConnection* instance sets the state to `LOCAL_CLOSING`. When a close frame is received, it yields a `CloseConnection` event, sets the state to `REMOTE_CLOSING` **and requires a reply to be sent**, this reply should be a `CloseConnection` event. To aid with this the `CloseConnection` class has a `response()` method to create the appropriate reply. For example,

```
if isinstance(event, CloseConnection):
    sock.send(ws.send(event.response()))
```

When the reply has been received by the initiator, it will also yield a `CloseConnection` event.

Regardless of which endpoint initiates the closing handshake, the server is responsible for tearing down the underlying connection. When the server receives a `CloseConnection` event, it should send pending *wsproto* data (if any) and then it can start tearing down the underlying connection.

Note: Both client and server connections must remember to reply to `CloseConnection` events initiated by the remote party.

2.6 Ping Pong

The *WSConnection* class supports sending WebSocket ping and pong frames via sending *Ping* and *Pong*. When a `Ping` frame is received it **requires a reply**, this reply should be a `Pong` event. To aid with this the `Ping` class has a `response()` method to create the appropriate reply. For example,

```
if isinstance(event, Ping):
    sock.send(ws.send(event.response()))
```

Note: Both client and server connections must remember to reply to `Ping` events initiated by the remote party.

CHAPTER 3

Advanced Usage

This document explains some of the more advanced usage concepts with *wsproto*. This is assume you are familiar with *wsproto* and I/O in Python.

3.1 Back-pressure

Back-pressure is an important concept to understand when implementing a client/server protocol. This section briefly explains the issue and then explains how to handle back-pressure when using *wsproto*.

Imagine that you have a WebSocket server that reads messages from the client, does some processing, and then sends a response. What happens if the client sends messages faster than the server can process them? If the incoming messages are buffered in memory, then the server will slowly use more and more memory, until the OS eventually kills it. This scenario is directly applicable to *wsproto*, because every time you call `receive_data(some_byte_string_of_data)`, it appends that data to an internal buffer.

The slow endpoint needs a way to signal the fast endpoint to stop sending messages until the slow endpoint can catch up. This signaling is called "back-pressure". As a Sans-IO library, *wsproto* is not responsible for network concerns like back-pressure, so that responsibility belongs to your network glue code.

Fortunately, TCP has the ability to signal backpressure, and the operating system will do that for you automatically—if you follow a few rules! The OS buffers all incoming and outgoing network data. Standard Python socket methods, such as `send(...)` and `recv()`, copy data to and from those OS buffers. For example, if the peer is sending data too quickly, then the OS receive buffer will start to get full, and the OS will signal the peer to stop transmitting. When `recv()` is called, the OS will copy data from its internal buffer into your process, free up space in its own buffer, and then signal to the peer to start transmitting again.

Therefore, you need to follow these two rules to implement back-pressure over TCP:

1. Do not receive from the socket faster than your code can process the messages. Your processing code may need to signal the receiving code when its ready to receive more data.
2. Do not store out-going messages in an unbounded collection. Ideally, out-going messages should be sent to the OS as soon as possible. If you need to buffer messages in memory, the buffer should be bounded so that it can not grow indefinitely.

3.2 Post handshake connection

A WebSocket connection starts with a handshake, which is an agreement to use the WebSocket protocol, and on which sub-protocol and extensions to use. It can be advantageous to perform this handshake outside of *wsproto*, for example in a dual stack setup whereby the HTTP handling is completed seperately. In this case the `Connection` class can be used directly.

```
connection = Connection(extensions)  # Agreed extensions
sock.send(connection.send(Message(data=b"Hi")))

connection.receive_data(sock.recv(4096))

for event in connection.events():
    # As with WSConnection, only without any handshake events
```

3.3 HTTP/2

WebSockets over HTTP/2 have a distinct difference to HTTP/1 in that only a single HTTP/2 stream is dedicated to the WebSocket rather than the entire connection (as in HTTP/1). This requires the HTTP/2 connection to be managed before the WebSocket connection with Hyper-h2 being recommended for HTTP/2.

Although *wsproto* doesn't manage the HTTP/2 connection it can still be used for the WebSocket stream. The HTTP/2 connection will need to handshake the WebSocket stream, with the key being agreement on the extensions used. Once the extensions have been agreed the `Connection` class can be used to manage the WebSocket connection, noting that data to be sent or received will need to be parsed by the HTTP/2 connection first. In practice for a server this looks like,

```
from wsproto.connection import Connection, ConnectionType
from wsproto.extensions import PerMessageDeflate
from wsproto.handshake import server_extensions_handshake

# WebSocket request has been received
request_extensions: List[str]
supported_extensions = [PerMessageDeflate()]
accepts = server_extensions_handshake(request_extensions, supported_extensions)
if accepts:
    response_headers.append({"sec-websocket-extensions": accepts})
# Send the response headers
connection = Connection(ConnectionType.SERVER, supported_extensions)
```

and for a client

```
from wsproto.connection import Connection, ConnectionType
from wsproto.extensions import PerMessageDeflate
from wsproto.handshake import client_extensions_handshake

# WebSocket response has been received
accepted_extensions: List[str]
proposed_extensions = [PerMessageDeflate()]
extensions = client_extensions_handshake(accepted_extensions, proposed_extensions)
connection = Connection(ConnectionType.CLIENT, supported_extensions)
```

any data received on the stream should be passed to the `connection` via the `receive_bytes` method and bytes returned from the `connection.send` method should be wrapped in a HTTP/2 data frame and sent.

CHAPTER 4

wsproto API

This document details the API of wsproto.

4.1 Semantic Versioning

wsproto 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 wsproto's public API. Anything not documented here is subject to change at any time.

4.2 Connection

class `wsproto.`**`WSConnection`**(*connection_type*)

4.3 Handshake

class `wsproto.handshake.`**`H11Handshake`**(*connection_type*)

A Handshake implementation for HTTP/1.1 connections.

`connection`

Return the established connection.

This will either return the connection or raise a LocalProtocolError if the connection has not yet been established.

`initiate_upgrade_connection`(*headers*, *path*)

Initiate an upgrade connection.

This should be used if the request has already be received and parsed.

receive_data(*data*)
Receive data from the remote.

A list of events that the remote peer triggered by sending this data can be retrieved with `events()`.

send(*event*)
Send an event to the remote.

This will return the bytes to send based on the event or raise a LocalProtocolError if the event is not valid given the state.

`wsproto.handshake.`**client_extensions_handshake**(*accepted*, *supported*)

`wsproto.handshake.`**server_extensions_handshake**(*requested*, *supported*)
Agree on the extensions to use returning an appropriate header value.

This returns None if there are no agreed extensions

4.4 Events

class `wsproto.events.`**Event**(***kwargs*)
Base class for wsproto events.

class `wsproto.events.`**Request**(***kwargs*)
The beginning of a Websocket connection, the HTTP Upgrade request

This event is fired when a SERVER connection receives a WebSocket handshake request (HTTP with upgrade header).

Fields:

extensions(*Union[List[Extension], List[str]]*)

extra_headers
The additional request headers, excluding extensions, host, subprotocols, and version headers.

host(*str*)
The hostname, or host header value.

subprotocols List[str]
A list of subprotocols ordered by preference.

target(*str*)
A list of the subprotocols proposed in the request, as a list of strings.

class `wsproto.events.`**AcceptConnection**(***kwargs*)
The acceptance of a Websocket upgrade request.

This event is fired when a CLIENT receives an acceptance response from a server. It is also used to accept an upgrade request when acting as a SERVER.

Fields:

class `wsproto.events.`**RejectConnection**(***kwargs*)
The rejection of a Websocket upgrade request, the HTTP response.

This event is fired when a CLIENT receives a rejection response from a server. It can be used to reject a request when sent from as SERVER. If has_body is False the headers must include a content-length or transfer encoding.

Fields:

headers(*List[Tuple[bytes, bytes]]*)
The headers to send with the response.

has_body
This defaults to False, but set to True if there is a body. See also *RejectData*.

status_code
The response status code.

class wsproto.events.**RejectData**(***kwargs*)
The rejection HTTP response body.

Fields:

body_finished
True if this is the final chunk of the body data.

data (*bytes*)
The raw body data.

class wsproto.events.**CloseConnection**(***kwargs*)
The end of a Websocket connection, represents a closure frame.

This event is fired after the connection is considered closed.

wsproto automatically emits a CLOSE frame when it receives one, to complete the close-handshake.

Fields:

code
The integer close code to indicate why the connection has closed.

reason
Additional reasoning for why the connection has closed.

class wsproto.events.**Message**(***kwargs*)
The websocket data message.

Fields:

data
The message data as byte string, can be decoded as UTF-8 for TEXT messages. This only represents a single chunk of data and not a full WebSocket message. You need to buffer and reassemble these chunks to get the full message.

frame_finished
This has no semantic content, but is provided just in case some weird edge case user wants to be able to reconstruct the fragmentation pattern of the original stream.

message_finished
True if this frame is the last one of this message, False if more frames are expected.

class wsproto.events.**TextMessage**(***kwargs*)
This event is fired when a data frame with TEXT payload is received.

class wsproto.events.**BytesMessage**(***kwargs*)
This event is fired when a data frame with BINARY payload is received.

class wsproto.events.**Ping**(***kwargs*)
The Ping event can be sent to trigger a ping frame and is fired when a Ping is received.

wsproto automatically emits a PONG frame with the same payload.

Fields:

payload
An optional payload to emit with the ping frame.

class `wsproto.events.`**`Pong`**`(**kwargs)`

The Pong event is fired when a Pong is received.

Fields:

`payload`

An optional payload to emit with the pong frame.

4.5 Frame Protocol

class `wsproto.frame_protocol.`**`Opcode`**

RFC 6455, Section 5.2 - Base Framing Protocol

`BINARY = 2`

Binary message

`CLOSE = 8`

Close frame

`CONTINUATION = 0`

Contiuation frame

`PING = 9`

Ping frame

`PONG = 10`

Pong frame

`TEXT = 1`

Text message

class `wsproto.frame_protocol.`**`CloseReason`**

RFC 6455, Section 7.4.1 - Defined Status Codes

`ABNORMAL_CLOSURE = 1006`

is a reserved value and MUST NOT be set as a status code in a Close control frame by an endpoint. It is designated for use in applications expecting a status code to indicate that the connection was closed abnormally, e.g., without sending or receiving a Close control frame.

`GOING_AWAY = 1001`

indicates that an endpoint is "going away", such as a server going down or a browser having navigated away from a page.

`INTERNAL_ERROR = 1011`

indicates that a server is terminating the connection because it encountered an unexpected condition that prevented it from fulfilling the request.

`INVALID_FRAME_PAYLOAD_DATA = 1007`

indicates that an endpoint is terminating the connection because it has received data within a message that was not consistent with the type of the message (e.g., non-UTF-8 [RFC3629] data within a text message).

`MANDATORY_EXT = 1010`

indicates that an endpoint (client) is terminating the connection because it has expected the server to negotiate one or more extension, but the server didn't return them in the response message of the WebSocket handshake. The list of extensions that are needed SHOULD appear in the /reason/ part of the Close frame. Note that this status code is not used by the server, because it can fail the WebSocket handshake instead.

`MESSAGE_TOO_BIG = 1009`

indicates that an endpoint is terminating the connection because it has received a message that is too big for it to process.

NORMAL_CLOSURE = 1000
indicates a normal closure, meaning that the purpose for which the connection was established has been fulfilled.

NO_STATUS_RCVD = 1005
is a reserved value and MUST NOT be set as a status code in a Close control frame by an endpoint. It is designated for use in applications expecting a status code to indicate that no status code was actually present.

POLICY_VIOLATION = 1008
indicates that an endpoint is terminating the connection because it has received a message that violates its policy. This is a generic status code that can be returned when there is no other more suitable status code (e.g., 1003 or 1009) or if there is a need to hide specific details about the policy.

PROTOCOL_ERROR = 1002
indicates that an endpoint is terminating the connection due to a protocol error.

SERVICE_RESTART = 1012
Server/service is restarting (not part of RFC6455)

TLS_HANDSHAKE_FAILED = 1015
is a reserved value and MUST NOT be set as a status code in a Close control frame by an endpoint. It is designated for use in applications expecting a status code to indicate that the connection was closed due to a failure to perform a TLS handshake (e.g., the server certificate can't be verified).

TRY_AGAIN_LATER = 1013
Temporary server condition forced blocking client's request (not part of RFC6455)

UNSUPPORTED_DATA = 1003
indicates that an endpoint is terminating the connection because it has received a type of data it cannot accept (e.g., an endpoint that understands only text data MAY send this if it receives a binary message).

4.6 Extensions

class wsproto.extensions.**Extension**

wsproto.extensions.**SUPPORTED_EXTENSIONS = {'permessage-deflate': <class 'wsproto.extensio**
SUPPORTED_EXTENSIONS maps all supported extension names to their class. This can be used to iterate all supported extensions of wsproto, instantiate new extensions based on their name, or check if a given extension is supported or not.

4.7 Exceptions

class wsproto.utilities.**LocalProtocolError**
Indicates an error due to local/programming errors.

This is raised when the connection is asked to do something that is either incompatible with the state or the websocket standard.

class wsproto.utilities.**RemoteProtocolError**(*message*, *event_hint=None*)
Indicates an error due to the remote's actions.

This is raised when processing the bytes from the remote if the remote has sent data that is incompatible with the websocket standard.

event_hint

This is a suggested wsproto Event to send to the client based on the error. It could be None if no hint is available.

Index