# wsproto Documentation

**Benno Rice**

**Dec 22, 2018**

# Contents

1 Installation 3

2 Getting Started 5
- 2.1 Connections . . . 5
- 2.2 Connecting to a WebSocket server . . . 6
- 2.3 WebSocket Servers . . . 7
- 2.4 Closing . . . 7
- 2.5 Ping Pong . . . 7
- 2.6 Back-pressure . . . 8

3 wsproto API 9
- 3.1 Semantic Versioning . . . 9
- 3.2 Connection . . . 9
- 3.3 Events . . . 10
- 3.4 Frame Protocol . . . 11
- 3.5 Extensions . . . 13

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_bytes`. When *wsproto* has data that needs to be sent over the network, you retrieve that data by calling `bytes_to_send`, and 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_bytes(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_data` method.

## 2.2 Connecting to a WebSocket server

Begin by instantiating a connection object. The `host` and `resource` arguments are required to instantiate a client. If the WebSocket server is located at `http://myhost.com/foo`, then you would instantiate the connection as follows:

```
ws = WSConnection(ConnectionType.CLIENT, host="myhost.com", resource='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:

```
stream = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
stream.connect(("myhost", 8000))
```

To read from the network:

```
data = stream.recv(4096)
ws.receive_bytes(data)
```

You also need to check if *wsproto* has data to send to the network:

```
data = ws.bytes_to_send()
stream.send(data)
```

Note that `bytes_to_send()` will return zero bytes if the protocol has no pending data. You can either poll this method or call it only when you expect to have pending data.

A standard Python socket will block on the call to `stream.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_data("Hello world!")
```

And to receive WebSocket events:

```
for event in ws.events():
    if isinstance(event, ConnectionEstablished):
        print('Connection established')
    elif isinstance(event, ConnectionClosed):
        print('Connection closed: code={} reason={}'.format(
            event.code, event.reason))
    elif isinstance(event, TextReceived):
        print('Received TEXT data: {}'.format(event.data))
        if event.message_finished:
            print('Message finished.')
    elif isinstance(event, BinaryReceived):
        print('Received BINARY data: {}'.format(event.data))
        if event.message_finished:
            print('BINARY Message finished.')
```

(continues on next page)

(continued from previous page)

```
    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 call the `accept` method after it receives a `ConnectionRequested` event:

```
for event in ws.events():
    if isinstance(event, ConnectionRequested):
        print('Accepting connection request')
        ws.accept(event)
    elif isinstance(event, ConnectionClosed):
        print('Connection closed: code={} reason={}'.format(
            event.code, event.reason))
    elif isinstance(event, TextReceived):
        print('Received TEXT data: {}'.format(event.data))
        if event.message_finished:
            print('TEXT Message finished.')
    elif isinstance(event, BinaryReceived):
        print('Received BINARY data: {}'.format(event.data))
        if event.message_finished:
            print('BINARY Message finished.')
    else:
        print('Unknown event: {!r}'.format(event))
```

For a more complete example, see synchronous_server.py.

## 2.4 Closing

WebSockets are closed with a handshake that requires each endpoint to send one frame and receive one frame. The `close()` method places a close frame in the send buffer. When a close frame is received, it yields a `ConnectionClosed` event, *and it also places a reply frame in the send buffer.* When that reply has been received by the initiator, it will also receive a `ConnectionClosed` event.

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

## 2.5 Ping Pong

The *`WSConnection`* class supports sending WebSocket ping and pong frames via the methods `ping` and `pong`.

**Note:** When a ping is received, *wsproto* automatically places a pong frame in its outgoing buffer. You should only call `pong()` if you want to send an unsolicited pong frame.

## 2.6 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 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_bytes()`, 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 like `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 buffere 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.

# CHAPTER 3

# wsproto API

This document details the API of wsproto.

## 3.1 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.

## 3.2 Connection

**class** `wsproto.connection.`**`WSConnection`**(*conn_type*)

A low-level WebSocket connection object.

This wraps two other protocol objects, an HTTP/1.1 protocol object used to do the initial HTTP upgrade handshake and a WebSocket frame protocol object used to exchange messages and other control frames.

**Parameters** **`conn_type`** (`ConnectionType`) – Whether this object is on the client- or server-side of a connection. To initialise as a client pass `CLIENT` otherwise pass `SERVER`.

**`bytes_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`

**events**()
Return a generator that provides any events that have been generated by protocol activity.

**Returns** generator of *Event* subclasses

**receive_bytes**(*data*)
Pass some received data to the connection for handling.

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

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

## 3.3 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**(*List[Extension]*)

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

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

**subprotocols**
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*)

**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.**Fail**(***kwargs*)
Indicates the connection handshake has failed.

**class** wsproto.events.**Data**(***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.

## 3.4 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).

## 3.5 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.

# Index


## A

ABNORMAL_CLOSURE (wsproto.frame_protocol.CloseReason attribute), 11
AcceptConnection (class in wsproto.events), 10

## B

BINARY (wsproto.frame_protocol.Opcode attribute), 11
bytes_to_send() (wsproto.connection.WSConnection method), 9
BytesMessage (class in wsproto.events), 11

## C

CLOSE (wsproto.frame_protocol.Opcode attribute), 11
CloseConnection (class in wsproto.events), 10
CloseReason (class in wsproto.frame_protocol), 11
code (wsproto.events.CloseConnection attribute), 10
CONTINUATION (wsproto.frame_protocol.Opcode attribute), 11

## D

Data (class in wsproto.events), 10
data (wsproto.events.Data attribute), 10

## E

Event (class in wsproto.events), 10
events() (wsproto.connection.WSConnection method), 9
Extension (class in wsproto.extensions), 13
extensions (wsproto.events.Request attribute), 10
extra_headers (wsproto.events.Request attribute), 10

## F

Fail (class in wsproto.events), 10
frame_finished (wsproto.events.Data attribute), 11

## G

GOING_AWAY (wsproto.frame_protocol.CloseReason attribute), 12

## H

host (wsproto.events.Request attribute), 10

## I

INTERNAL_ERROR (wsproto.frame_protocol.CloseReason attribute), 12
INVALID_FRAME_PAYLOAD_DATA (wsproto.frame_protocol.CloseReason attribute), 12

## M

MANDATORY_EXT (wsproto.frame_protocol.CloseReason attribute), 12
message_finished (wsproto.events.Data attribute), 11
MESSAGE_TOO_BIG (wsproto.frame_protocol.CloseReason attribute), 12

## N

NO_STATUS_RCVD (wsproto.frame_protocol.CloseReason attribute), 12
NORMAL_CLOSURE (wsproto.frame_protocol.CloseReason attribute), 12

## O

Opcode (class in wsproto.frame_protocol), 11

## P

payload (wsproto.events.Ping attribute), 11
payload (wsproto.events.Pong attribute), 11
Ping (class in wsproto.events), 11
PING (wsproto.frame_protocol.Opcode attribute), 11
POLICY_VIOLATION (wsproto.frame_protocol.CloseReason attribute), 12
Pong (class in wsproto.events), 11
PONG (wsproto.frame_protocol.Opcode attribute), 11
PROTOCOL_ERROR (wsproto.frame_protocol.CloseReason attribute), 12

## R

reason (wsproto.events.CloseConnection attribute), 10
receive_bytes() (wsproto.connection.WSConnection method), 10
Request (class in wsproto.events), 10

## S

SERVICE_RESTART (wsproto.frame_protocol.CloseReason attribute), 12
subprotocols (wsproto.events.Request attribute), 10
SUPPORTED_EXTENSIONS (in module wsproto.extensions), 13

## T

target (wsproto.events.Request attribute), 10
TEXT (wsproto.frame_protocol.Opcode attribute), 11
TextMessage (class in wsproto.events), 11
TLS_HANDSHAKE_FAILED (wsproto.frame_protocol.CloseReason attribute), 12
TRY_AGAIN_LATER (wsproto.frame_protocol.CloseReason attribute), 12

## U

UNSUPPORTED_DATA (wsproto.frame_protocol.CloseReason attribute), 12

## W

WSConnection (class in wsproto.connection), 9