websockets Documentation

Release 8.0.1

Aymeric Augustin

Jul 21, 2019

Contents

websockets is a library for building WebSocket servers and clients in Python with a focus on correctness and simplicity.

Built on top of asyncio, Python's standard asynchronous I/O framework, it provides an elegant coroutine-based API.

Here's how a client sends and receives messages:

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

import asyncio
import websockets

async def hello():
    uri = "ws://localhost:8765"
    async with websockets.connect(uri) as websocket:
        await websocket.send("Hello world!")
        await websocket.recv()

asyncio.get_event_loop().run_until_complete(hello())
```

And here's an echo server:

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

import asyncio
import websockets

async def echo(websocket, path):
    async for message in websocket:
        await websocket.send(message)

start_server = websockets.serve(echo, "localhost", 8765)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Do you like it? Let's dive in!

CHAPTER 1

Tutorials

If you're new to websockets, this is the place to start.

1.1 Getting started

1.1.1 Requirements

websockets requires Python 3.6.

You should use the latest version of Python if possible. If you're using an older version, be aware that for each minor version (3.x), only the latest bugfix release (3.x.y) is officially supported.

1.1.2 Installation

Install websockets with:

```
pip install websockets
```

1.1.3 Basic example

Here's a WebSocket server example.

It reads a name from the client, sends a greeting, and closes the connection.

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

# WS server example

import asyncio
import websockets
```

(continues on next page)

(continued from previous page)

```
async def hello(websocket, path):
    name = await websocket.recv()
    print(f"< {name}")

    greeting = f"Hello {name}!"

    await websocket.send(greeting)
    print(f"> {greeting}")

start_server = websockets.serve(hello, "localhost", 8765)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

On the server side, `websockets` executes the handler coroutine `hello` once for each WebSocket connection. It closes the connection when the handler coroutine returns.

Here's a corresponding WebSocket client example.

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

# WS client example

import asyncio
import websockets

async def hello():
    uri = "ws://localhost:8765"
    async with websockets.connect(uri) as websocket:
        name = input("What's your name? ")

        await websocket.send(name)
        print(f"> {name}")

        greeting = await websocket.recv()
        print(f"< {greeting}")

asyncio.get_event_loop().run_until_complete(hello())
```

Using `connect()` as an asynchronous context manager ensures the connection is closed before exiting the `hello` coroutine.

1.1.4 Secure example

Secure WebSocket connections improve confidentiality and also reliability because they reduce the risk of interference by bad proxies.

The WSS protocol is to WS what HTTPS is to HTTP: the connection is encrypted with Transport Layer Security (TLS) — which is often referred to as Secure Sockets Layer (SSL). WSS requires TLS certificates like HTTPS.

Here's how to adapt the server example to provide secure connections. See the documentation of the `ssl` module for configuring the context securely.

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

(continues on next page)

(continued from previous page)

```
# WSS (WS over TLS) server example, with a self-signed certificate

import asyncio
import pathlib
import ssl
import websockets

async def hello(websocket, path):
    name = await websocket.recv()
    print(f"< {name}")

    greeting = f"Hello {name}!"

    await websocket.send(greeting)
    print(f"> {greeting}")

ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
localhost_pem = pathlib.Path(__file__).with_name("localhost.pem")
ssl_context.load_cert_chain(localhost_pem)

start_server = websockets.serve(
    hello, "localhost", 8765, ssl=ssl_context
)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Here's how to adapt the client.

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

# WSS (WS over TLS) client example, with a self-signed certificate

import asyncio
import pathlib
import ssl
import websockets

ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
localhost_pem = pathlib.Path(__file__).with_name("localhost.pem")
ssl_context.load_verify_locations(localhost_pem)

async def hello():
    uri = "wss://localhost:8765"
    async with websockets.connect(
        uri, ssl=ssl_context
    ) as websocket:
        name = input("What's your name? ")

        await websocket.send(name)
        print(f"> {name}")

        greeting = await websocket.recv()
        print(f"< {greeting}")

asyncio.get_event_loop().run_until_complete(hello())
```

This client needs a context because the server uses a self-signed certificate.

A client connecting to a secure WebSocket server with a valid certificate (i.e. signed by a CA that your Python installation trusts) can simply pass `ssl=True` to `connect()` instead of building a context.

1.1.5 Browser-based example

Here's an example of how to run a WebSocket server and connect from a browser.

Run this script in a console:

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

# WS server that sends messages at random intervals

import asyncio
import datetime
import random
import websockets

async def time(websocket, path):
    while True:
        now = datetime.datetime.utcnow().isoformat() + "Z"
        await websocket.send(now)
        await asyncio.sleep(random.random() * 3)

start_server = websockets.serve(time, "127.0.0.1", 5678)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Then open this HTML file in a browser.

```
<!DOCTYPE html>
<html>
    <head>
        <title>WebSocket demo</title>
    </head>
    <body>
        <script>
            var ws = new WebSocket("ws://127.0.0.1:5678/"),
                messages = document.createElement('ul');
            ws.onmessage = function (event) {
                var messages = document.getElementsByTagName('ul')[0],
                    message = document.createElement('li'),
                    content = document.createTextNode(event.data);
                message.appendChild(content);
                messages.appendChild(message);
            };
            document.body.appendChild(messages);
        </script>
    </body>
</html>
```

1.1.6 Synchronization example

A WebSocket server can receive events from clients, process them to update the application state, and synchronize the resulting state across clients.

Here's an example where any client can increment or decrement a counter. Updates are propagated to all connected clients.

The concurrency model of `asyncio` guarantees that updates are serialized.

Run this script in a console:

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

# WS server example that synchronizes state across clients

import asyncio
import json
import logging
import websockets

logging.basicConfig()

STATE = {"value": 0}

USERS = set()


def state_event():
    return json.dumps({"type": "state", **STATE})


def users_event():
    return json.dumps({"type": "users", "count": len(USERS)})


async def notify_state():
    if USERS:  # asyncio.wait doesn't accept an empty list
        message = state_event()
        await asyncio.wait([user.send(message) for user in USERS])


async def notify_users():
    if USERS:  # asyncio.wait doesn't accept an empty list
        message = users_event()
        await asyncio.wait([user.send(message) for user in USERS])


async def register(websocket):
    USERS.add(websocket)
    await notify_users()


async def unregister(websocket):
    USERS.remove(websocket)
    await notify_users()


async def counter(websocket, path):
```

(continues on next page)

(continued from previous page)

```
    # register(websocket) sends user_event() to websocket
    await register(websocket)
    try:
        await websocket.send(state_event())
        async for message in websocket:
            data = json.loads(message)
            if data["action"] == "minus":
                STATE["value"] -= 1
                await notify_state()
            elif data["action"] == "plus":
                STATE["value"] += 1
                await notify_state()
            else:
                logging.error("unsupported event: {}", data)
    finally:
        await unregister(websocket)


start_server = websockets.serve(counter, "localhost", 6789)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Then open this HTML file in several browsers.

```
<!DOCTYPE html>
<html>
    <head>
        <title>WebSocket demo</title>
        <style type="text/css">
            body {
                font-family: "Courier New", sans-serif;
                text-align: center;
            }
            .buttons {
                font-size: 4em;
                display: flex;
                justify-content: center;
            }
            .button, .value {
                line-height: 1;
                padding: 2rem;
                margin: 2rem;
                border: medium solid;
                min-height: 1em;
                min-width: 1em;
            }
            .button {
                cursor: pointer;
                user-select: none;
            }
            .minus {
                color: red;
            }
            .plus {
                color: green;
            }
```

(continues on next page)

(continued from previous page)

```
            .value {
                min-width: 2em;
            }
            .state {
                font-size: 2em;
            }
        </style>
    </head>
    <body>
        <div class="buttons">
            <div class="minus button">-</div>
            <div class="value">?</div>
            <div class="plus button">+</div>
        </div>
        <div class="state">
            <span class="users">?</span> online
        </div>
        <script>
            var minus = document.querySelector('.minus'),
                plus = document.querySelector('.plus'),
                value = document.querySelector('.value'),
                users = document.querySelector('.users'),
                websocket = new WebSocket("ws://127.0.0.1:6789/");
            minus.onclick = function (event) {
                websocket.send(JSON.stringify({action: 'minus'}));
            }
            plus.onclick = function (event) {
                websocket.send(JSON.stringify({action: 'plus'}));
            }
            websocket.onmessage = function (event) {
                data = JSON.parse(event.data);
                switch (data.type) {
                    case 'state':
                        value.textContent = data.value;
                        break;
                    case 'users':
                        users.textContent = (
                            data.count.toString() + " user" +
                            (data.count == 1 ? "" : "s"));
                        break;
                    default:
                        console.error(
                            "unsupported event", data);
                }
            };
        </script>
    </body>
</html>
```

1.1.7 Common patterns

You will usually want to process several messages during the lifetime of a connection. Therefore you must write a loop. Here are the basic patterns for building a WebSocket server.

Consumer

For receiving messages and passing them to a `consumer` coroutine:

```
async def consumer_handler(websocket, path):
    async for message in websocket:
        await consumer(message)
```

In this example, `consumer` represents your business logic for processing messages received on the WebSocket connection.

Iteration terminates when the client disconnects.

Producer

For getting messages from a `producer` coroutine and sending them:

```
async def producer_handler(websocket, path):
    while True:
        message = await producer()
        await websocket.send(message)
```

In this example, `producer` represents your business logic for generating messages to send on the WebSocket connection.

`send()` raises a *`ConnectionClosed`* exception when the client disconnects, which breaks out of the `while True` loop.

Both

You can read and write messages on the same connection by combining the two patterns shown above and running the two tasks in parallel:

```
async def handler(websocket, path):
    consumer_task = asyncio.ensure_future(
        consumer_handler(websocket, path))
    producer_task = asyncio.ensure_future(
        producer_handler(websocket, path))
    done, pending = await asyncio.wait(
        [consumer_task, producer_task],
        return_when=asyncio.FIRST_COMPLETED,
    )
    for task in pending:
        task.cancel()
```

Registration

As shown in the synchronization example above, if you need to maintain a list of currently connected clients, you must register them when they connect and unregister them when they disconnect.

```
connected = set()

async def handler(websocket, path):
    # Register.
```

(continues on next page)

(continued from previous page)

```
    connected.add(websocket)
    try:
        # Implement logic here.
        await asyncio.wait([ws.send("Hello!") for ws in connected])
        await asyncio.sleep(10)
    finally:
        # Unregister.
        connected.remove(websocket)
```

This simplistic example keeps track of connected clients in memory. This only works as long as you run a single process. In a practical application, the handler may subscribe to some channels on a message broker, for example.

1.1.8 That's all!

The design of the `websockets` API was driven by simplicity.

You don't have to worry about performing the opening or the closing handshake, answering pings, or any other behavior required by the specification.

`websockets` handles all this under the hood so you don't have to.

1.1.9 One more thing…

`websockets` provides an interactive client:

```
$ python -m websockets wss://echo.websocket.org/
```

1.2 FAQ

Note: Many questions asked in `websockets`' issue tracker are actually about `asyncio`. Python's documentation about developing with asyncio is a good complement.

1.2.1 Server side

Why does the server close the connection after processing one message?

Your connection handler exits after processing one message. Write a loop to process multiple messages.

For example, if your handler looks like this:

```
async def handler(websocket, path):
    print(websocket.recv())
```

change it like this:

```
async def handler(websocket, path):
    async for message in websocket:
        print(message)
```

Don't feel bad if this happens to you — it's the most common question in websockets' issue tracker :-)

Why can only one client connect at a time?

Your connection handler blocks the event loop. Look for blocking calls. Any call that may take some time must be asynchronous.

For example, if you have:

```
async def handler(websocket, path):
    time.sleep(1)
```

change it to:

```
async def handler(websocket, path):
    await asyncio.sleep(1)
```

This is part of learning asyncio. It isn't specific to websockets.

See also Python's documentation about running blocking code.

How do I get access HTTP headers, for example cookies?

To access HTTP headers during the WebSocket handshake, you can override *process_request*:

```
async def process_request(self, path, request_headers):
    cookies = request_header["Cookie"]
```

See

Once the connection is established, they're available in `request_headers`:

```
async def handler(websocket, path):
    cookies = websocket.request_headers["Cookie"]
```

How do I get the IP address of the client connecting to my server?

It's available in *remote_address*:

```
async def handler(websocket, path):
    remote_ip = websocket.remote_address[0]
```

How do I set which IP addresses my server listens to?

Look at the `host` argument of `create_server()`.

`serve()` accepts the same arguments as `create_server()`.

How do I close a connection properly?

websockets takes care of closing the connection when the handler exits.

How do I run a HTTP server and WebSocket server on the same port?

This isn't supported.

Providing a HTTP server is out of scope for websockets. It only aims at providing a WebSocket server.

There's limited support for returning HTTP responses with the *process_request* hook. If you need more, pick a HTTP server and run it separately.

1.2.2 Client side

How do I close a connection properly?

The easiest is to use `connect()` as a context manager:

```
async with connect(...) as websocket:
    ...
```

How do I reconnect automatically when the connection drops?

See issue 414.

How do I disable TLS/SSL certificate verification?

Look at the `ssl` argument of `create_connection()`.

`connect()` accepts the same arguments as `create_connection()`.

1.2.3 Both sides

How do I do two things in parallel? How do I integrate with another coroutine?

You must start two tasks, which the event loop will run concurrently. You can achieve this with `asyncio.gather()` or `asyncio.wait()`.

This is also part of learning asyncio and not specific to websockets.

Keep track of the tasks and make sure they terminate or you cancel them when the connection terminates.

How do I create channels or topics?

websockets doesn't have built-in publish / subscribe for these use cases.

Depending on the scale of your service, a simple in-memory implementation may do the job or you may need an external publish / subscribe component.

What does `ConnectionClosedError: code = 1006` mean?

If you're seeing this traceback in the logs of a server:

```
Error in connection handler
Traceback (most recent call last):
  ...
asyncio.streams.IncompleteReadError: 0 bytes read on a total of 2 expected bytes

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  ...
websockets.exceptions.ConnectionClosedError: code = 1006 (connection closed␣
↪abnormally [internal]), no reason
```

or if a client crashes with this traceback:

```
Traceback (most recent call last):
  ...
ConnectionResetError: [Errno 54] Connection reset by peer

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  ...
websockets.exceptions.ConnectionClosedError: code = 1006 (connection closed␣
↪abnormally [internal]), no reason
```

it means that the TCP connection was lost. As a consequence, the WebSocket connection was closed without receiving a close frame, which is abnormal.

You can catch and handle `ConnectionClosed` to prevent it from being logged.

There are several reasons why long-lived connections may be lost:

- End-user devices tend to lose network connectivity often and unpredictably because they can move out of wireless network coverage, get unplugged from a wired network, enter airplane mode, be put to sleep, etc.
- HTTP load balancers or proxies that aren't configured for long-lived connections may terminate connections after a short amount of time, usually 30 seconds.

If you're facing a reproducible issue, *enable debug logs* to see when and how connections are closed.

Are there `onopen`, `onmessage`, `onerror`, and `onclose` callbacks?

No, there aren't.

websockets provides high-level, coroutine-based APIs. Compared to callbacks, coroutines make it easier to manage control flow in concurrent code.

If you prefer callback-based APIs, you should use another library.

Can I use `websockets` synchronously, without `async` / `await`?

You can convert every asynchronous call to a synchronous call by wrapping it in `asyncio.get_event_loop().run_until_complete(...)`.

If this turns out to be impractical, you should use another library.

1.2.4 Miscellaneous

How do I set a timeout on `recv()`?

Use `wait_for()`:

```
await asyncio.wait_for(websocket.recv(), timeout=10)
```

This technique works for most APIs, except for asynchronous context managers. See issue 574.

How do I keep idle connections open?

websockets sends pings at 20 seconds intervals to keep the connection open.

In closes the connection if it doesn't get a pong within 20 seconds.

You can adjust this behavior with `ping_interval` and `ping_timeout`.

How do I respond to pings?

websockets takes care of responding to pings with pongs.

Is there a Python 2 version?

No, there isn't.

websockets builds upon asyncio which requires Python 3.

CHAPTER 2

How-to guides

These guides will help you build and deploy a `websockets` application.

2.1 Cheat sheet

2.1.1 Server

- Write a coroutine that handles a single connection. It receives a WebSocket protocol instance and the URI path in argument.
  - Call `recv()` and `send()` to receive and send messages at any time.
  - When `recv()` or `send()` raises `ConnectionClosed`, clean up and exit. If you started other `asyncio.Task`, terminate them before exiting.
  - If you aren't awaiting `recv()`, consider awaiting `wait_closed()` to detect quickly when the connection is closed.
  - You may `ping()` or `pong()` if you wish but it isn't needed in general.
- Create a server with `serve()` which is similar to asyncio's `create_server()`. You can also use it as an asynchronous context manager.
  - The server takes care of establishing connections, then lets the handler execute the application logic, and finally closes the connection after the handler exits normally or with an exception.
  - For advanced customization, you may subclass `WebSocketServerProtocol` and pass either this subclass or a factory function as the `create_protocol` argument.

2.1.2 Client

- Create a client with `connect()` which is similar to asyncio's `create_connection()`. You can also use it as an asynchronous context manager.

   - For advanced customization, you may subclass WebSocketClientProtocol and pass either this subclass or a factory function as the create_protocol argument.

- Call *recv()* and *send()* to receive and send messages at any time.
- You may *ping()* or *pong()* if you wish but it isn't needed in general.
- If you aren't using *connect()* as a context manager, call *close()* to terminate the connection.

2.1.3 Debugging

If you don't understand what websockets is doing, enable logging:

```
import logging
logger = logging.getLogger('websockets')
logger.setLevel(logging.INFO)
logger.addHandler(logging.StreamHandler())
```

The logs contain:

- Exceptions in the connection handler at the ERROR level
- Exceptions in the opening or closing handshake at the INFO level
- All frames at the DEBUG level — this can be very verbose

If you're new to asyncio, you will certainly encounter issues that are related to asynchronous programming in general rather than to websockets in particular. Fortunately Python's official documentation provides advice to develop with asyncio. Check it out: it's invaluable!

2.1.4 Passing additional arguments to the connection handler

When writing a server, if you need to pass additional arguments to the connection handler, you can bind them with functools.partial():

```
import asyncio
import functools
import websockets

async def handler(websocket, path, extra_argument):
    ...

bound_handler = functools.partial(handler, extra_argument='spam')
start_server = websockets.serve(bound_handler, '127.0.0.1', 8765)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Another way to achieve this result is to define the handler coroutine in a scope where the extra_argument variable exists instead of injecting it through an argument.

2.2 Deployment

2.2.1 Application server

The author of `websockets` isn't aware of best practices for deploying network services based on `asyncio`, let alone application servers.

You can run a script similar to the *server example*, inside a supervisor if you deem that useful.

You can also add a wrapper to daemonize the process. Third-party libraries provide solutions for that.

If you can share knowledge on this topic, please file an issue. Thanks!

2.2.2 Graceful shutdown

You may want to close connections gracefully when shutting down the server, perhaps after executing some cleanup logic. There are two ways to achieve this with the object returned by `serve()`:

- using it as a asynchronous context manager, or
- calling its `close()` method, then waiting for its `wait_closed()` method to complete.

On Unix systems, shutdown is usually triggered by sending a signal.

Here's a full example (Unix-only):

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

import asyncio
import signal
import websockets

async def echo(websocket, path):
    async for message in websocket:
        await websocket.send(message)

async def echo_server(stop):
    async with websockets.serve(echo, "localhost", 8765):
        await stop

loop = asyncio.get_event_loop()

# The stop condition is set when receiving SIGTERM.
stop = loop.create_future()
loop.add_signal_handler(signal.SIGTERM, stop.set_result, None)

# Run the server until the stop condition is met.
loop.run_until_complete(echo_server(stop))
```

It's more difficult to achieve the same effect on Windows. Some third-party projects try to help with this problem.

If your server doesn't run in the main thread, look at `call_soon_threadsafe()`.

2.2.3 Memory usage

In most cases, memory usage of a WebSocket server is proportional to the number of open connections. When a server handles thousands of connections, memory usage can become a bottleneck.

Memory usage of a single connection is the sum of:

1. the baseline amount of memory `websockets` requires for each connection,
2. the amount of data held in buffers before the application processes it,
3. any additional memory allocated by the application itself.

Baseline

Compression settings are the main factor affecting the baseline amount of memory used by each connection.

By default `websockets` maximizes compression rate at the expense of memory usage. If memory usage is an issue, lowering compression settings can help:

- Context Takeover is necessary to get good performance for almost all applications. It should remain enabled.
- Window Bits is a trade-off between memory usage and compression rate. It defaults to 15 and can be lowered. The default value isn't optimal for small, repetitive messages which are typical of WebSocket servers.
- Memory Level is a trade-off between memory usage and compression speed. It defaults to 8 and can be lowered. A lower memory level can actually increase speed thanks to memory locality, even if the CPU does more work!

See this *example* for how to configure compression settings.

Here's how various compression settings affect memory usage of a single connection on a 64-bit system, as well a benchmark of compressed size and compression time for a corpus of small JSON documents.

Compression	Window Bits	Memory Level	Memory usage	Size vs. default	Time vs. default
default	15	8	325 KiB	+0%	+0%
	14	7	181 KiB	+1.5%	-5.3%
	13	6	110 KiB	+2.8%	-7.5%
	12	5	73 KiB	+4.4%	-18.9%
	11	4	55 KiB	+8.5%	-18.8%
disabled	N/A	N/A	22 KiB	N/A	N/A

Don't assume this example is representative! Compressed size and compression time depend heavily on the kind of messages exchanged by the application!

You can run the same benchmark for your application by creating a list of typical messages and passing it to the `_benchmark` function.

This blog post by Ilya Grigorik provides more details about how compression settings affect memory usage and how to optimize them.

This experiment by Peter Thorson suggests Window Bits = 11, Memory Level = 4 as a sweet spot for optimizing memory usage.

Buffers

Under normal circumstances, buffers are almost always empty.

Under high load, if a server receives more messages than it can process, bufferbloat can result in excessive memory use.

By default `websockets` has generous limits. It is strongly recommended to adapt them to your application. When you call `serve()`:

- Set max_size (default: 1 MiB, UTF-8 encoded) to the maximum size of messages your application generates.
- Set max_queue (default: 32) to the maximum number of messages your application expects to receive faster than it can process them. The queue provides burst tolerance without slowing down the TCP connection.

Furthermore, you can lower read_limit and write_limit (default: 64 KiB) to reduce the size of buffers for incoming and outgoing data.

The design document provides *more details about buffers*.

2.2.4 Port sharing

The WebSocket protocol is an extension of HTTP/1.1. It can be tempting to serve both HTTP and WebSocket on the same port.

The author of websockets doesn't think that's a good idea, due to the widely different operational characteristics of HTTP and WebSocket.

websockets provide minimal support for responding to HTTP requests with the *process_request()* hook. Typical use cases include health checks. Here's an example:

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

# WS echo server with HTTP endpoint at /health/

import asyncio
import http
import websockets

async def health_check(path, request_headers):
    if path == "/health/":
        return http.HTTPStatus.OK, [], b"OK\n"

async def echo(websocket, path):
    async for message in websocket:
        await websocket.send(message)

start_server = websockets.serve(
    echo, "localhost", 8765, process_request=health_check
)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

2.3 Extensions

The WebSocket protocol supports extensions.

At the time of writing, there's only one registered extension, WebSocket Per-Message Deflate, specified in **RFC 7692**.

2.3.1 Per-Message Deflate

serve() and *connect()* enable the Per-Message Deflate extension by default. You can disable this with compression=None.

You can also configure the Per-Message Deflate extension explicitly if you want to customize its parameters.

Here's an example on the server side:

```
import websockets
from websockets.extensions import permessage_deflate

websockets.serve(
    ...,
    extensions=[
        permessage_deflate.ServerPerMessageDeflateFactory(
            server_max_window_bits=11,
            client_max_window_bits=11,
            compress_settings={'memLevel': 4},
        ),
    ],
)
```

Here's an example on the client side:

```
import websockets
from websockets.extensions import permessage_deflate

websockets.connect(
    ...,
    extensions=[
        permessage_deflate.ClientPerMessageDeflateFactory(
            server_max_window_bits=11,
            client_max_window_bits=11,
            compress_settings={'memLevel': 4},
        ),
    ],
)
```

Refer to the API documentation of *ServerPerMessageDeflateFactory* and *ClientPerMessageDeflateFactory* for details.

2.3.2 Writing an extension

During the opening handshake, WebSocket clients and servers negotiate which extensions will be used with which parameters. Then each frame is processed by extensions before it's sent and after it's received.

As a consequence writing an extension requires implementing several classes:

1. Extension Factory: it negotiates parameters and instantiates the extension. Clients and servers require separate extension factories with distinct APIs.
2. Extension: it decodes incoming frames and encodes outgoing frames. If the extension is symmetrical, clients and servers can use the same class.

`websockets` provides abstract base classes for extension factories and extensions.

class websockets.extensions.base.**ServerExtensionFactory**
Abstract class for server-side extension factories.

name
Extension identifier.

Return type `str`

process_request_params (*params*, *accepted_extensions*)
Process request parameters received from the client.

To accept the offer, return a 2-uple containing:

- response parameters: a list of `(name, value)` pairs
- an extension: an instance of a subclass of *Extension*

Parameters

- **params** (Sequence[Tuple[str, Optional[str]]]) – list of `(name, value)` pairs.
- **accepted_extensions** (Sequence[*Extension*]) – list of previously accepted extensions.

Raises ***NegotiationError*** – to reject the offer, if parameters aren't acceptable

Return type Tuple[List[Tuple[str, Optional[str]]], *Extension*]

class websockets.extensions.base.**ClientExtensionFactory**

Abstract class for client-side extension factories.

get_request_params()

Build request parameters.

Return a list of `(name, value)` pairs.

Return type List[Tuple[str, Optional[str]]]

name

Extension identifier.

Return type str

process_response_params(*params*, *accepted_extensions*)

Process response parameters received from the server.

Parameters

- **params** (Sequence[Tuple[str, Optional[str]]]) – list of `(name, value)` pairs.
- **accepted_extensions** (Sequence[*Extension*]) – list of previously accepted extensions.

Raises ***NegotiationError*** – if parameters aren't acceptable

Return type *Extension*

class websockets.extensions.base.**Extension**

Abstract class for extensions.

decode(*frame*, *, *max_size=None*)

Decode an incoming frame.

Parameters

- **frame** (*Frame*) – incoming frame
- **max_size** (Optional[int]) – maximum payload size in bytes

Return type *Frame*

encode(*frame*)

Encode an outgoing frame.

Parameters **frame** (*Frame*) – outgoing frame

Return type *Frame*

name

Extension identifier.

Return type str

CHAPTER 3

Reference

Find all the details you could ask for, and then some.

3.1 API

3.1.1 Design

`websockets` provides complete client and server implementations, as shown in the *getting started guide*. These functions are built on top of low-level APIs reflecting the two phases of the WebSocket protocol:

1. An opening handshake, in the form of an HTTP Upgrade request;
2. Data transfer, as framed messages, ending with a closing handshake.

The first phase is designed to integrate with existing HTTP software. `websockets` provides a minimal implementation to build, parse and validate HTTP requests and responses.

The second phase is the core of the WebSocket protocol. `websockets` provides a complete implementation on top of `asyncio` with a simple API.

For convenience, public APIs can be imported directly from the `websockets` package, unless noted otherwise. Anything that isn't listed in this document is a private API.

3.1.2 High-level

Server

`websockets.server` defines the WebSocket server APIs.

await websockets.server.**serve**(*ws_handler*, *host=None*, *port=None*, ***, *create_protocol=None*, *ping_interval=20*, *ping_timeout=20*, *close_timeout=10*, *max_size=2 ** 20*, *max_queue=2 ** 5*, *read_limit=2 ** 16*, *write_limit=2 ** 16*, *loop=None*, *compression='deflate'*, *origins=None*, *extensions=None*, *subprotocols=None*, *extra_headers=None*, *process_request=None*, *select_subprotocol=None*, ***kwds*)

Create, start, and return a WebSocket server on `host` and `port`.

Whenever a client connects, the server accepts the connection, creates a `WebSocketServerProtocol`, performs the opening handshake, and delegates to the connection handler defined by `ws_handler`. Once the handler completes, either normally or with an exception, the server performs the closing handshake and closes the connection.

Awaiting `serve()` yields a `WebSocketServer`. This instance provides `close()` and `wait_closed()` methods for terminating the server and cleaning up its resources.

When a server is closed with `close()`, it closes all connections with close code 1001 (going away). Connections handlers, which are running the `ws_handler` coroutine, will receive a `ConnectionClosedOK` exception on their current or next interaction with the WebSocket connection.

`serve()` can also be used as an asynchronous context manager. In this case, the server is shut down when exiting the context.

`serve()` is a wrapper around the event loop's `create_server()` method. It creates and starts a `Server` with `create_server()`. Then it wraps the `Server` in a `WebSocketServer` and returns the `WebSocketServer`.

The `ws_handler` argument is the WebSocket handler. It must be a coroutine accepting two arguments: a `WebSocketServerProtocol` and the request URI.

The `host` and `port` arguments, as well as unrecognized keyword arguments, are passed along to `create_server()`.

For example, you can set the `ssl` keyword argument to a `SSLContext` to enable TLS.

The `create_protocol` parameter allows customizing the `Protocol` that manages the connection. It should be a callable or class accepting the same arguments as `WebSocketServerProtocol` and returning an instance of `WebSocketServerProtocol` or a subclass. It defaults to `WebSocketServerProtocol`.

The behavior of `ping_interval`, `ping_timeout`, `close_timeout`, `max_size`, `max_queue`, `read_limit`, and `write_limit` is described in `WebSocketCommonProtocol`.

`serve()` also accepts the following optional arguments:

- `compression` is a shortcut to configure compression extensions; by default it enables the "permessage-deflate" extension; set it to `None` to disable compression
- `origins` defines acceptable Origin HTTP headers; include `None` if the lack of an origin is acceptable
- `extensions` is a list of supported extensions in order of decreasing preference
- `subprotocols` is a list of supported subprotocols in order of decreasing preference
- `extra_headers` sets additional HTTP response headers when the handshake succeeds; it can be a `Headers` instance, a `Mapping`, an iterable of `(name, value)` pairs, or a callable taking the request path and headers in arguments and returning one of the above
- `process_request` allows intercepting the HTTP request; it must be a coroutine taking the request path and headers in argument; see `process_request()` for details

- `select_subprotocol` allows customizing the logic for selecting a subprotocol; it must be a callable taking the subprotocols offered by the client and available on the server in argument; see *select_subprotocol()* for details

Since there's no useful way to propagate exceptions triggered in handlers, they're sent to the `'websockets.server'` logger instead. Debugging is much easier if you configure logging to print them:

```
import logging
logger = logging.getLogger('websockets.server')
logger.setLevel(logging.ERROR)
logger.addHandler(logging.StreamHandler())
```

await websockets.server.**unix_serve**(*ws_handler*, *path*, ***kwargs*)

Similar to *serve()*, but for listening on Unix sockets.

This function calls the event loop's `create_unix_server()` method.

It is only available on Unix.

It's useful for deploying a server behind a reverse proxy such as nginx.

Parameters **path** (`str`) – file system path to the Unix socket

Return type `Serve`

class websockets.server.**WebSocketServer**(*loop*)

WebSocket server returned by *serve()*.

This class provides the same interface as `AbstractServer`, namely the `close()` and `wait_closed()` methods.

It keeps track of WebSocket connections in order to close them properly when shutting down.

Instances of this class store a reference to the `Server` object returned by `create_server()` rather than inherit from `Server` in part because `create_server()` doesn't support passing a custom `Server` class.

close()

Close the server.

This method:

- closes the underlying `Server`;
- rejects new WebSocket connections with an HTTP 503 (service unavailable) error; this happens when the server accepted the TCP connection but didn't complete the WebSocket opening handshake prior to closing;
- closes open WebSocket connections with close code 1001 (going away).

close() is idempotent.

Return type `None`

await wait_closed()

Wait until the server is closed.

When *wait_closed()* returns, all TCP connections are closed and all connection handlers have returned.

Return type `None`

sockets

List of `socket` objects the server is listening to.

`None` if the server is closed.

Return type Optional[List[socket]]

class websockets.server.**WebSocketServerProtocol**(*ws_handler*, *ws_server*, ***, *origins=None*, *extensions=None*, *subprotocols=None*, *extra_headers=None*, *process_request=None*, *select_subprotocol=None*, ***kwargs*)

Protocol subclass implementing a WebSocket server.

This class inherits most of its methods from *WebSocketCommonProtocol*.

For the sake of simplicity, it doesn't rely on a full HTTP implementation. Its support for HTTP responses is very limited.

await handshake(*origins=None*, *available_extensions=None*, *available_subprotocols=None*, *extra_headers=None*)

Perform the server side of the opening handshake.

Return the path of the URI of the request.

Parameters

- **origins** (Optional[Sequence[Optional[NewType()(Origin, str)]]]) – list of acceptable values of the Origin HTTP header; include None if the lack of an origin is acceptable
- **available_extensions** (Optional[Sequence[*ServerExtensionFactory*]]) – list of supported extensions in the order in which they should be used
- **available_subprotocols** (Optional[Sequence[NewType()(Subprotocol, str)]]) – list of supported subprotocols in order of decreasing preference
- **extra_headers** (Union[*Headers*, Mapping[str, str], Iterable[Tuple[str, str]], Callable[[str, *Headers*], Union[*Headers*, Mapping[str, str], Iterable[Tuple[str, str]]]], None]) – sets additional HTTP response headers when the handshake succeeds; it can be a *Headers* instance, a Mapping, an iterable of (name, value) pairs, or a callable taking the request path and headers in arguments and returning one of the above.

Raises ***InvalidHandshake*** – if the handshake fails

Return type str

await process_request(*path*, *request_headers*)

Intercept the HTTP request and return an HTTP response if appropriate.

If process_request returns None, the WebSocket handshake continues. If it returns 3-uple containing a status code, response headers and a response body, that HTTP response is sent and the connection is closed. In that case:

- The HTTP status must be a HTTPStatus.
- HTTP headers must be a *Headers* instance, a Mapping, or an iterable of (name, value) pairs.
- The HTTP response body must be bytes. It may be empty.

This coroutine may be overridden in a *WebSocketServerProtocol* subclass, for example:

- to return a HTTP 200 OK response on a given path; then a load balancer can use this path for a health check;
- to authenticate the request and return a HTTP 401 Unauthorized or a HTTP 403 Forbidden when authentication fails.

Instead of subclassing, it is possible to override this method by passing a `process_request` argument to the *serve()* function or the *WebSocketServerProtocol* constructor. This is equivalent, except `process_request` won't have access to the protocol instance, so it can't store information for later use.

`process_request` is expected to complete quickly. If it may run for a long time, then it should await `wait_closed()` and exit if `wait_closed()` completes, or else it could prevent the server from shutting down.

Parameters

- **path** (`str`) – request path, including optional query string
- **request_headers** (*Headers*) – request headers

Return type `Optional[Tuple[HTTPStatus, Union[Headers, Mapping[str, str], Iterable[Tuple[str, str]]], bytes]]`

select_subprotocol (*client_subprotocols*, *server_subprotocols*)

Pick a subprotocol among those offered by the client.

If several subprotocols are supported by the client and the server, the default implementation selects the preferred subprotocols by giving equal value to the priorities of the client and the server.

If no subprotocol is supported by the client and the server, it proceeds without a subprotocol.

This is unlikely to be the most useful implementation in practice, as many servers providing a subprotocol will require that the client uses that subprotocol. Such rules can be implemented in a subclass.

Instead of subclassing, it is possible to override this method by passing a `select_subprotocol` argument to the *serve()* function or the *WebSocketServerProtocol* constructor

Parameters

- **client_subprotocols** (`Sequence[NewType()(Subprotocol, str)]`) – list of subprotocols offered by the client
- **server_subprotocols** (`Sequence[NewType()(Subprotocol, str)]`) – list of subprotocols available on the server

Return type `Optional[NewType()(Subprotocol, str)]`

Client

websockets.client defines the WebSocket client APIs.

await websockets.client.**connect** (*uri*, **, *create_protocol=None*, *ping_interval=20*, *ping_timeout=20*, *close_timeout=10*, *max_size=2 ** 20*, *max_queue=2 ** 5*, *read_limit=2 ** 16*, *write_limit=2 ** 16*, *loop=None*, *compression='deflate'*, *origin=None*, *extensions=None*, *subprotocols=None*, *extra_headers=None*, ***kwds*)

Connect to the WebSocket server at the given `uri`.

Awaiting *connect()* yields a *WebSocketClientProtocol* which can then be used to send and receive messages.

connect() can also be used as a asynchronous context manager. In that case, the connection is closed when exiting the context.

connect() is a wrapper around the event loop's `create_connection()` method. Unknown keyword arguments are passed to `create_connection()`.

For example, you can set the `ssl` keyword argument to a `SSLContext` to enforce some TLS settings. When connecting to a `wss://` URI, if this argument isn't provided explicitly, it's set to `True`, which means Python's default `SSLContext` is used.

You can connect to a different host and port from those found in `uri` by setting `host` and `port` keyword arguments. This only changes the destination of the TCP connection. The host name from `uri` is still used in the TLS handshake for secure connections and in the `Host` HTTP header.

The `create_protocol` parameter allows customizing the `Protocol` that manages the connection. It should be a callable or class accepting the same arguments as `WebSocketClientProtocol` and returning an instance of `WebSocketClientProtocol` or a subclass. It defaults to `WebSocketClientProtocol`.

The behavior of `ping_interval`, `ping_timeout`, `close_timeout`, `max_size`, `max_queue`, `read_limit`, and `write_limit` is described in `WebSocketCommonProtocol`.

`connect()` also accepts the following optional arguments:

- `compression` is a shortcut to configure compression extensions; by default it enables the "permessage-deflate" extension; set it to `None` to disable compression
- `origin` sets the Origin HTTP header
- `extensions` is a list of supported extensions in order of decreasing preference
- `subprotocols` is a list of supported subprotocols in order of decreasing preference
- `extra_headers` sets additional HTTP request headers; it can be a `Headers` instance, a `Mapping`, or an iterable of `(name, value)` pairs

Raises

- **InvalidURI** – if `uri` is invalid
- **InvalidHandshake** – if the opening handshake fails

await websockets.client.**unix_connect**(*path*, *uri='ws://localhost/'*, ***kwargs*)

Similar to `connect()`, but for connecting to a Unix socket.

This function calls the event loop's `create_unix_connection()` method.

It is only available on Unix.

It's mainly useful for debugging servers listening on Unix sockets.

Parameters

- **path** (`str`) – file system path to the Unix socket
- **uri** (`str`) – WebSocket URI

Return type `Connect`

class websockets.client.**WebSocketClientProtocol**(***, *origin=None*, *extensions=None*, *subprotocols=None*, *extra_headers=None*, ***kwargs*)

`Protocol` subclass implementing a WebSocket client.

This class inherits most of its methods from `WebSocketCommonProtocol`.

await handshake(*wsuri*, *origin=None*, *available_extensions=None*, *available_subprotocols=None*, *extra_headers=None*)

Perform the client side of the opening handshake.

Parameters

- **origin** (`Optional[NewType()(Origin, str)]`) – sets the Origin HTTP header

- **available_extensions** (Optional[Sequence[*ClientExtensionFactory*]]) – list of supported extensions in the order in which they should be used
- **available_subprotocols** (Optional[Sequence[NewType()(Subprotocol, str)]]) – list of supported subprotocols in order of decreasing preference
- **extra_headers** (Union[*Headers*, Mapping[str, str], Iterable[Tuple[str, str]], None]) – sets additional HTTP request headers; it must be a *Headers* instance, a Mapping, or an iterable of (name, value) pairs

Raises ***InvalidHandshake*** – if the handshake fails

Return type None

Shared

websockets.protocol handles WebSocket control and data frames.

See sections 4 to 8 of RFC 6455.

class websockets.protocol.**WebSocketCommonProtocol**(*, *ping_interval=20*, *ping_timeout=20*, *close_timeout=None*, *max_size=1048576*, *max_queue=32*, *read_limit=65536*, *write_limit=65536*, *loop=None*, *host=None*, *port=None*, *secure=None*, *legacy_recv=False*, *timeout=None*)

Protocol subclass implementing the data transfer phase.

Once the WebSocket connection is established, during the data transfer phase, the protocol is almost symmetrical between the server side and the client side. *WebSocketCommonProtocol* implements logic that's shared between servers and clients..

Subclasses such as *WebSocketServerProtocol* and *WebSocketClientProtocol* implement the opening handshake, which is different between servers and clients.

WebSocketCommonProtocol performs four functions:

- It runs a task that stores incoming data frames in a queue and makes them available with the *recv()* coroutine.
- It sends outgoing data frames with the *send()* coroutine.
- It deals with control frames automatically.
- It performs the closing handshake.

WebSocketCommonProtocol supports asynchronous iteration:

```
async for message in websocket:
    await process(message)
```

The iterator yields incoming messages. It exits normally when the connection is closed with the close code 1000 (OK) or 1001 (going away). It raises a *ConnectionClosedError* exception when the connection is closed with any other code.

Once the connection is open, a Ping frame is sent every ping_interval seconds. This serves as a keepalive. It helps keeping the connection open, especially in the presence of proxies with short timeouts on inactive connections. Set ping_interval to None to disable this behavior.

If the corresponding Pong frame isn't received within ping_timeout seconds, the connection is considered unusable and is closed with code 1011. This ensures that the remote endpoint remains responsive. Set ping_timeout to None to disable this behavior.

The close_timeout parameter defines a maximum wait time in seconds for completing the closing handshake and terminating the TCP connection. *close()* completes in at most 4 * close_timeout on the server side and 5 * close_timeout on the client side.

close_timeout needs to be a parameter of the protocol because websockets usually calls *close()* implicitly:

- on the server side, when the connection handler terminates,
- on the client side, when exiting the context manager for the connection.

To apply a timeout to any other API, wrap it in wait_for().

The max_size parameter enforces the maximum size for incoming messages in bytes. The default value is 1 MiB. None disables the limit. If a message larger than the maximum size is received, *recv()* will raise *ConnectionClosedError* and the connection will be closed with code 1009.

The max_queue parameter sets the maximum length of the queue that holds incoming messages. The default value is 32. None disables the limit. Messages are added to an in-memory queue when they're received; then *recv()* pops from that queue. In order to prevent excessive memory consumption when messages are received faster than they can be processed, the queue must be bounded. If the queue fills up, the protocol stops processing incoming data until *recv()* is called. In this situation, various receive buffers (at least in asyncio and in the OS) will fill up, then the TCP receive window will shrink, slowing down transmission to avoid packet loss.

Since Python can use up to 4 bytes of memory to represent a single character, each connection may use up to 4 * max_size * max_queue bytes of memory to store incoming messages. By default, this is 128 MiB. You may want to lower the limits, depending on your application's requirements.

The read_limit argument sets the high-water limit of the buffer for incoming bytes. The low-water limit is half the high-water limit. The default value is 64 KiB, half of asyncio's default (based on the current implementation of StreamReader).

The write_limit argument sets the high-water limit of the buffer for outgoing bytes. The low-water limit is a quarter of the high-water limit. The default value is 64 KiB, equal to asyncio's default (based on the current implementation of FlowControlMixin).

As soon as the HTTP request and response in the opening handshake are processed:

- the request path is available in the path attribute;
- the request and response HTTP headers are available in the request_headers and response_headers attributes, which are *Headers* instances.

If a subprotocol was negotiated, it's available in the subprotocol attribute.

Once the connection is closed, the code is available in the close_code attribute and the reason in close_reason.

All these attributes must be treated as read-only.

await close(*code=1000*, *reason="*)

Perform the closing handshake.

close() waits for the other end to complete the handshake and for the TCP connection to terminate. As a consequence, there's no need to await *wait_closed()*; *close()* already does it.

close() is idempotent: it doesn't do anything once the connection is closed.

Wrapping *close()* in create_task() is safe, given that errors during connection termination aren't particularly useful.

Canceling `close()` is discouraged. If it takes too long, you can set a shorter `close_timeout`. If you don't want to wait, let the Python process exit, then the OS will close the TCP connection.

Parameters

- **code** (`int`) – WebSocket close code
- **reason** (`str`) – WebSocket close reason

Return type `None`

await wait_closed()

Wait until the connection is closed.

This is identical to `closed`, except it can be awaited.

This can make it easier to handle connection termination, regardless of its cause, in tasks that interact with the WebSocket connection.

Return type `None`

await recv()

Receive the next message.

Return a `str` for a text frame and `bytes` for a binary frame.

When the end of the message stream is reached, `recv()` raises `ConnectionClosed`. Specifically, it raises `ConnectionClosedOK` after a normal connection closure and `ConnectionClosedError` after a protocol error or a network failure.

Changed in version 3.0: `recv()` used to return `None` instead. Refer to the changelog for details.

Canceling `recv()` is safe. There's no risk of losing the next message. The next invocation of `recv()` will return it. This makes it possible to enforce a timeout by wrapping `recv()` in `wait_for()`.

Raises

- **ConnectionClosed** – when the connection is closed
- **RuntimeError** – if two coroutines call `recv()` concurrently

Return type `Union[str, bytes]`

await send(*message*)

Send a message.

A string (`str`) is sent as a Text frame. A bytestring or bytes-like object (`bytes`, `bytearray`, or `memoryview`) is sent as a Binary frame.

`send()` also accepts an iterable or an asynchronous iterable of strings, bytestrings, or bytes-like objects. In that case the message is fragmented. Each item is treated as a message fragment and sent in its own frame. All items must be of the same type, or else `send()` will raise a `TypeError` and the connection will be closed.

Canceling `send()` is discouraged. Instead, you should close the connection with `close()`. Indeed, there only two situations where `send()` yields control to the event loop:

1. The write buffer is full. If you don't want to wait until enough data is sent, your only alternative is to close the connection. `close()` will likely time out then abort the TCP connection.
2. `message` is an asynchronous iterator. Stopping in the middle of a fragmented message will cause a protocol error. Closing the connection has the same effect.

Raises **TypeError** – for unsupported inputs

Return type `None`

await ping(*data=None*)
Send a ping.

Return a Future which will be completed when the corresponding pong is received and which you may ignore if you don't want to wait.

A ping may serve as a keepalive or as a check that the remote endpoint received all messages up to this point:

```
pong_waiter = await ws.ping()
await pong_waiter   # only if you want to wait for the pong
```

By default, the ping contains four random bytes. This payload may be overridden with the optional `data` argument which must be a string (which will be encoded to UTF-8) or a bytes-like object.

Canceling *ping()* is discouraged. If *ping()* doesn't return immediately, it means the write buffer is full. If you don't want to wait, you should close the connection.

Canceling the Future returned by *ping()* has no effect.

Return type Awaitable[None]

await pong(*data=b”*)
Send a pong.

An unsolicited pong may serve as a unidirectional heartbeat.

The payload may be set with the optional `data` argument which must be a string (which will be encoded to UTF-8) or a bytes-like object.

Canceling *pong()* is discouraged for the same reason as *ping()*.

Return type None

local_address
Local address of the connection.

This is a `(host, port)` tuple or `None` if the connection hasn't been established yet.

Return type Any

remote_address
Remote address of the connection.

This is a `(host, port)` tuple or `None` if the connection hasn't been established yet.

Return type Any

open
`True` when the connection is usable.

It may be used to detect disconnections. However, this approach is discouraged per the EAFP principle.

When `open` is `False`, using the connection raises a *ConnectionClosed* exception.

Return type bool

closed
`True` once the connection is closed.

Be aware that both *open* and *closed* are `False` during the opening and closing sequences.

Return type bool

Types

`websockets.typing.Data = typing.Union[str, bytes]`

Union type; Union[X, Y] means either X or Y.

To define a union, use e.g. Union[int, str]. Details:

- The arguments must be types and there must be at least one.
- None as an argument is a special case and is replaced by type(None).
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int  # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str]
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- When two arguments have a subclass relationship, the least derived argument is kept, e.g.:

```
class Employee: pass
class Manager(Employee): pass
Union[int, Employee, Manager] == Union[int, Employee]
Union[Manager, int, Employee] == Union[int, Employee]
Union[Employee, Manager] == Employee
```

- Similar for object:

```
Union[int, object] == object
```

- You cannot subclass or instantiate a union.
- You can use Optional[X] as a shorthand for Union[X, None].

Per-Message Deflate Extension

websockets.extensions.permessage_deflate implements the Compression Extensions for WebSocket as specified in **RFC 7692**.

class websockets.extensions.permessage_deflate.**ServerPerMessageDeflateFactory**(*server_no_context_*
*client_no_context_*
server_max_windo
client_max_windo
com-
press_settings=No

Server-side extension factory for the Per-Message Deflate extension.

Parameters behave as described in section 7.1 of RFC 7692. Set them to `True` to include them in the negotiation offer without a value or to an integer value to include them with this value.

Parameters

- **server_no_context_takeover** (bool) – defaults to False
- **client_no_context_takeover** (bool) – defaults to False
- **server_max_window_bits** (Optional[int]) – optional, defaults to None
- **client_max_window_bits** (Optional[int]) – optional, defaults to None
- **compress_settings** (Optional[Dict[str, Any]]) – optional, keyword arguments for zlib.compressobj(), excluding wbits

class websockets.extensions.permessage_deflate.**ClientPerMessageDeflateFactory**(*server_no_context client_no_context_ server_max_windo client_max_windo com- press_settings=No*

Client-side extension factory for the Per-Message Deflate extension.

Parameters behave as described in section 7.1 of RFC 7692. Set them to True to include them in the negotiation offer without a value or to an integer value to include them with this value.

Parameters

- **server_no_context_takeover** (bool) – defaults to False
- **client_no_context_takeover** (bool) – defaults to False
- **server_max_window_bits** (Optional[int]) – optional, defaults to None
- **client_max_window_bits** (Optional[int]) – optional, defaults to None
- **compress_settings** (Optional[Dict[str, Any]]) – optional, keyword arguments for zlib.compressobj(), excluding wbits

HTTP Basic Auth

websockets.auth provides HTTP Basic Authentication according to **RFC 7235** and **RFC 7617**.

websockets.auth.**basic_auth_protocol_factory**(*realm, credentials=None, check_credentials=None, create_protocol=<class 'websockets.auth.BasicAuthWebSocketServerProtocol'>*)

Protocol factory that enforces HTTP Basic Auth.

basic_auth_protocol_factory is designed to integrate with *serve()* like this:

```
websockets.serve(
    ...,
    create_protocol=websockets.basic_auth_protocol_factory(
        realm="my dev server",
        credentials=("hello", "iloveyou"),
    )
)
```

realm indicates the scope of protection. It should contain only ASCII characters because the encoding of non-ASCII characters is undefined. Refer to section 2.2 of **RFC 7235** for details.

credentials defines hard coded authorized credentials. It can be a (username, password) pair or a list of such pairs.

`check_credentials` defines a coroutine that checks whether credentials are authorized. This coroutine receives `username` and `password` arguments and returns a `bool`.

One of `credentials` or `check_credentials` must be provided but not both.

By default, `basic_auth_protocol_factory` creates a factory for building *`BasicAuthWebSocketServerProtocol`* instances. You can override this with the `create_protocol` parameter.

Parameters

- **`realm`** (`str`) – scope of protection
- **`credentials`** (`Union[Tuple[str, str], Iterable[Tuple[str, str]], None]`) – hard coded credentials
- **`check_credentials`** (`Optional[Callable[[str, str], Awaitable[bool]]]`) – coroutine that verifies credentials

Raises **`TypeError`** – if the credentials argument has the wrong type

Return type `Callable[[Any], BasicAuthWebSocketServerProtocol]`

class `websockets.auth.`**`BasicAuthWebSocketServerProtocol`**(**args, realm, check_credentials, **kwargs*)

WebSocket server protocol that enforces HTTP Basic Auth.

await **`process_request`**(*path, request_headers*)

Check HTTP Basic Auth and return a HTTP 401 or 403 response if needed.

If authentication succeeds, the username of the authenticated user is stored in the `username` attribute.

Return type `Optional[Tuple[HTTPStatus, Union[Headers, Mapping[str, str], Iterable[Tuple[str, str]]], bytes]]`

Exceptions

`websockets.exceptions` defines the following exception hierarchy:

- ***`WebSocketException`***
  - ***`ConnectionClosed`***
    - *`ConnectionClosedError`*
    - *`ConnectionClosedOK`*
  - ***`InvalidHandshake`***
    - *`SecurityError`*
    - *`InvalidMessage`*
    - ***`InvalidHeader`***
      - *`InvalidHeaderFormat`*
      - *`InvalidHeaderValue`*
      - *`InvalidOrigin`*
      - *`InvalidUpgrade`*
    - *`InvalidStatusCode`*

        * ***NegotiationError***
            · *DuplicateParameter*
            · *InvalidParameterName*
            · *InvalidParameterValue*
        * *AbortHandshake*
        * *RedirectHandshake*
    – *InvalidState*
    – *InvalidURI*
    – *PayloadTooBig*
    – *ProtocolError*

exception websockets.exceptions.**WebSocketException**
Base class for all exceptions defined by websockets.

exception websockets.exceptions.**ConnectionClosed**(*code*, *reason*)
Raised when trying to interact with a closed connection.

Provides the connection close code and reason in its code and reason attributes respectively.

exception websockets.exceptions.**ConnectionClosedError**(*code*, *reason*)
Like *ConnectionClosed*, when the connection terminated with an error.

This means the close code is different from 1000 (OK) and 1001 (going away).

exception websockets.exceptions.**ConnectionClosedOK**(*code*, *reason*)
Like *ConnectionClosed*, when the connection terminated properly.

This means the close code is 1000 (OK) or 1001 (going away).

exception websockets.exceptions.**InvalidHandshake**
Raised during the handshake when the WebSocket connection fails.

exception websockets.exceptions.**SecurityError**
Raised when a handshake request or response breaks a security rule.

Security limits are hard coded.

exception websockets.exceptions.**InvalidMessage**
Raised when a handshake request or response is malformed.

exception websockets.exceptions.**InvalidHeader**(*name*, *value=None*)
Raised when a HTTP header doesn't have a valid format or value.

exception websockets.exceptions.**InvalidHeaderFormat**(*name*, *error*, *header*, *pos*)
Raised when a HTTP header cannot be parsed.

The format of the header doesn't match the grammar for that header.

exception websockets.exceptions.**InvalidHeaderValue**(*name*, *value=None*)
Raised when a HTTP header has a wrong value.

The format of the header is correct but a value isn't acceptable.

exception websockets.exceptions.**InvalidOrigin**(*origin*)
Raised when the Origin header in a request isn't allowed.

exception websockets.exceptions.**InvalidUpgrade**(*name*, *value=None*)
Raised when the Upgrade or Connection header isn't correct.

exception websockets.exceptions.**InvalidStatusCode**(*status_code*)
Raised when a handshake response status code is invalid.

The integer status code is available in the `status_code` attribute.

exception websockets.exceptions.**NegotiationError**
Raised when negotiating an extension fails.

exception websockets.exceptions.**DuplicateParameter**(*name*)
Raised when a parameter name is repeated in an extension header.

exception websockets.exceptions.**InvalidParameterName**(*name*)
Raised when a parameter name in an extension header is invalid.

exception websockets.exceptions.**InvalidParameterValue**(*name*, *value*)
Raised when a parameter value in an extension header is invalid.

exception websockets.exceptions.**AbortHandshake**(*status*, *headers*, *body=b''*)
Raised to abort the handshake on purpose and return a HTTP response.

This exception is an implementation detail.

The public API is `process_request()`.

exception websockets.exceptions.**RedirectHandshake**(*uri*)
Raised when a handshake gets redirected.

This exception is an implementation detail.

exception websockets.exceptions.**InvalidState**
Raised when an operation is forbidden in the current state.

This exception is an implementation detail.

It should never be raised in normal circumstances.

exception websockets.exceptions.**InvalidURI**(*uri*)
Raised when connecting to an URI that isn't a valid WebSocket URI.

exception websockets.exceptions.**PayloadTooBig**
Raised when receiving a frame with a payload exceeding the maximum size.

exception websockets.exceptions.**ProtocolError**
Raised when the other side breaks the protocol.

websockets.exceptions.**WebSocketProtocolError**
alias of *websockets.exceptions.ProtocolError*

3.1.3 Low-level

Opening handshake

websockets.handshake provides helpers for the WebSocket handshake.

See section 4 of RFC 6455.

Some checks cannot be performed because they depend too much on the context; instead, they're documented below.

To accept a connection, a server must:

- Read the request, check that the method is GET, and check the headers with *check_request()*,
- Send a 101 response to the client with the headers created by *build_response()* if the request is valid; otherwise, send an appropriate HTTP error code.

To open a connection, a client must:

- Send a GET request to the server with the headers created by *build_request()*,
- Read the response, check that the status code is 101, and check the headers with *check_response()*.

websockets.handshake.**build_request**(*headers*)

Build a handshake request to send to the server.

Update request headers passed in argument.

Parameters **headers** (*Headers*) – request headers

Return type str

Returns key which must be passed to *check_response()*

websockets.handshake.**check_request**(*headers*)

Check a handshake request received from the client.

This function doesn't verify that the request is an HTTP/1.1 or higher GET request and doesn't perform Host and Origin checks. These controls are usually performed earlier in the HTTP request handling code. They're the responsibility of the caller.

Parameters **headers** (*Headers*) – request headers

Return type str

Returns key which must be passed to *build_response()*

Raises ***InvalidHandshake*** – if the handshake request is invalid; then the server must return 400 Bad Request error

websockets.handshake.**build_response**(*headers*, *key*)

Build a handshake response to send to the client.

Update response headers passed in argument.

Parameters

- **headers** (*Headers*) – response headers
- **key** (str) – comes from *check_request()*

Return type None

websockets.handshake.**check_response**(*headers*, *key*)

Check a handshake response received from the server.

This function doesn't verify that the response is an HTTP/1.1 or higher response with a 101 status code. These controls are the responsibility of the caller.

Parameters

- **headers** (*Headers*) – response headers
- **key** (str) – comes from *build_request()*

Raises ***InvalidHandshake*** – if the handshake response is invalid

Return type None

Data transfer

websockets.framing reads and writes WebSocket frames.

It deals with a single frame at a time. Anything that depends on the sequence of frames is implemented in *websockets.protocol*.

See section 5 of RFC 6455.

class websockets.framing.**Frame**

WebSocket frame.

Parameters

- **fin** (*bool*) – FIN bit
- **rsv1** (*bool*) – RSV1 bit
- **rsv2** (*bool*) – RSV2 bit
- **rsv3** (*bool*) – RSV3 bit
- **opcode** (*int*) – opcode
- **data** (*bytes*) – payload data

Only these fields are needed. The MASK bit, payload length and masking-key are handled on the fly by *read()* and *write()*.

check()

Check that reserved bits and opcode have acceptable values.

Raises ***ProtocolError*** – if a reserved bit or the opcode is invalid

Return type None

classmethod await read(*reader*, ***, *mask*, *max_size=None*, *extensions=None*)

Read a WebSocket frame.

Parameters

- **reader** (Callable[[int], Awaitable[bytes]]) – coroutine that reads exactly the requested number of bytes, unless the end of file is reached
- **mask** (bool) – whether the frame should be masked i.e. whether the read happens on the server side
- **max_size** (Optional[int]) – maximum payload size in bytes
- **extensions** (Optional[Sequence[*Extension*]]) – list of classes with a decode() method that transforms the frame and return a new frame; extensions are applied in reverse order

Raises

- ***PayloadTooBig*** – if the frame exceeds max_size
- ***ProtocolError*** – if the frame contains incorrect values

Return type *Frame*

write(*writer*, ***, *mask*, *extensions=None*)

Write a WebSocket frame.

Parameters

- **frame** – frame to write

- **writer** (Callable[[bytes], Any]) – function that writes bytes
- **mask** (bool) – whether the frame should be masked i.e. whether the write happens on the client side
- **extensions** (Optional[Sequence[*Extension*]]) – list of classes with an encode() method that transform the frame and return a new frame; extensions are applied in order

Raises *ProtocolError* – if the frame contains incorrect values

Return type None

websockets.framing.**prepare_data**(*data*)
Convert a string or byte-like object to an opcode and a bytes-like object.

This function is designed for data frames.

If data is a str, return OP_TEXT and a bytes object encoding data in UTF-8.

If data is a bytes-like object, return OP_BINARY and a bytes-like object.

Raises **TypeError** – if data doesn't have a supported type

Return type Tuple[int, bytes]

websockets.framing.**encode_data**(*data*)
Convert a string or byte-like object to bytes.

This function is designed for ping and pong frames.

If data is a str, return a bytes object encoding data in UTF-8.

If data is a bytes-like object, return a bytes object.

Raises **TypeError** – if data doesn't have a supported type

Return type bytes

websockets.framing.**parse_close**(*data*)
Parse the payload from a close frame.

Return (code, reason).

Raises

- *ProtocolError* – if data is ill-formed
- **UnicodeDecodeError** – if the reason isn't valid UTF-8

Return type Tuple[int, str]

websockets.framing.**serialize_close**(*code*, *reason*)
Serialize the payload for a close frame.

This is the reverse of *parse_close()*.

Return type bytes

URI parser

websockets.uri parses WebSocket URIs.

See section 3 of RFC 6455.

websockets.uri.**parse_uri**(*uri*)
Parse and validate a WebSocket URI.

Raises **ValueError** – if uri isn't a valid WebSocket URI.

Return type *WebSocketURI*

class websockets.uri.**WebSocketURI**

WebSocket URI.

Parameters

- **secure** (*bool*) – secure flag
- **host** (*str*) – lower-case host
- **port** (*int*) – port, always set even if it's the default
- **resource_name** (*str*) – path and optional query
- **user_info** (*str*) – (username, password) tuple when the URI contains User Information, else None.

Utilities

websockets.headers provides parsers and serializers for HTTP headers used in WebSocket handshake messages.

These APIs cannot be imported from websockets. They must be imported from *websockets.headers*.

websockets.headers.**parse_connection**(*header*)

Parse a Connection header.

Return a list of HTTP connection options.

Parameters **header** (str) – value of the Connection header

Raises ***InvalidHeaderFormat*** – on invalid inputs.

Return type List[NewType()(ConnectionOption, str)]

websockets.headers.**parse_upgrade**(*header*)

Parse an Upgrade header.

Return a list of HTTP protocols.

Parameters **header** (str) – value of the Upgrade header

Raises ***InvalidHeaderFormat*** – on invalid inputs.

Return type List[NewType()(UpgradeProtocol, str)]

websockets.headers.**parse_extension**(*header*)

Parse a Sec-WebSocket-Extensions header.

Return a list of WebSocket extensions and their parameters in this format:

```
[
    (
        'extension name',
        [
            ('parameter name', 'parameter value'),
            ....
        ]
    ),
    ...
]
```

Parameter values are None when no value is provided.

Raises *InvalidHeaderFormat* – on invalid inputs.

Return type List[Tuple[str, List[Tuple[str, Optional[str]]]]]

websockets.headers.**build_extension**(*extensions*)
Build a Sec-WebSocket-Extensions header.

This is the reverse of *parse_extension()*.

Return type str

websockets.headers.**parse_subprotocol**(*header*)
Parse a Sec-WebSocket-Protocol header.

Return a list of WebSocket subprotocols.

Raises *InvalidHeaderFormat* – on invalid inputs.

Return type List[NewType()(Subprotocol, str)]

websockets.headers.**build_subprotocol**(*protocols*)
Build a Sec-WebSocket-Protocol header.

This is the reverse of *parse_subprotocol()*.

Return type str

websockets.headers.**build_www_authenticate_basic**(*realm*)
Build a WWW-Authenticate header for HTTP Basic Auth.

Parameters **realm** (str) – authentication realm

Return type str

websockets.headers.**parse_authorization_basic**(*header*)
Parse an Authorization header for HTTP Basic Auth.

Return a (username, password) tuple.

Parameters **header** (str) – value of the Authorization header

Raises

- *InvalidHeaderFormat* – on invalid inputs
- *InvalidHeaderValue* – on unsupported inputs

Return type Tuple[str, str]

websockets.headers.**build_authorization_basic**(*username*, *password*)
Build an Authorization header for HTTP Basic Auth.

This is the reverse of *parse_authorization_basic()*.

Return type str

websockets.http module provides basic HTTP/1.1 support. It is merely :adequate for WebSocket handshake messages.

These APIs cannot be imported from websockets. They must be imported from *websockets.http*.

await websockets.http.**read_request**(*stream*)
Read an HTTP/1.1 GET request and returns (path, headers).

path isn't URL-decoded or validated in any way.

path and headers are expected to contain only ASCII characters. Other characters are represented with surrogate escapes.

read_request() doesn't attempt to read the request body because WebSocket handshake requests don't have one. If the request contains a body, it may be read from `stream` after this coroutine returns.

Parameters **stream** (`StreamReader`) – input to read the request from

Raises

- **EOFError** – if the connection is closed without a full HTTP request
- ***SecurityError*** – if the request exceeds a security limit
- **ValueError** – if the request isn't well formatted

Return type `Tuple[str, Headers]`

await websockets.http.**read_response**(*stream*)

Read an HTTP/1.1 response and returns `(status_code, reason, headers)`.

`reason` and `headers` are expected to contain only ASCII characters. Other characters are represented with surrogate escapes.

read_request() doesn't attempt to read the response body because WebSocket handshake responses don't have one. If the response contains a body, it may be read from `stream` after this coroutine returns.

Parameters **stream** (`StreamReader`) – input to read the response from

Raises

- **EOFError** – if the connection is closed without a full HTTP response
- ***SecurityError*** – if the response exceeds a security limit
- **ValueError** – if the response isn't well formatted

Return type `Tuple[int, str, Headers]`

class websockets.http.**Headers**(**args*, ***kwargs*)

Efficient data structure for manipulating HTTP headers.

A `list` of `(name, values)` is inefficient for lookups.

A `dict` doesn't suffice because header names are case-insensitive and multiple occurrences of headers with the same name are possible.

Headers stores HTTP headers in a hybrid data structure to provide efficient insertions and lookups while preserving the original data.

In order to account for multiple values with minimal hassle, *Headers* follows this logic:

- **When getting a header with `headers[name]`:**
  - if there's no value, `KeyError` is raised;
  - if there's exactly one value, it's returned;
  - if there's more than one value, *MultipleValuesError* is raised.
- When setting a header with `headers[name] = value`, the value is appended to the list of values for that header.
- When deleting a header with `del headers[name]`, all values for that header are removed (this is slow).

Other methods for manipulating headers are consistent with this logic.

As long as no header occurs multiple times, *Headers* behaves like `dict`, except keys are lower-cased to provide case-insensitivity.

Two methods support support manipulating multiple values explicitly:

- `get_all()` returns a list of all values for a header;
- `raw_items()` returns an iterator of `(name, values)` pairs.

clear()

Remove all headers.

Return type None

get_all(*key*)

Return the (possibly empty) list of all values for a header.

Parameters **key** (str) – header name

Return type List[str]

raw_items()

Return an iterator of all values as `(name, value)` pairs.

Return type Iterator[Tuple[str, str]]

exception websockets.http.**MultipleValuesError**

Exception raised when `Headers` has more than one value for a key.

CHAPTER 4

Discussions

Get a deeper understanding of how `websockets` is built and why.

4.1 Design

This document describes the design of `websockets`. It assumes familiarity with the specification of the WebSocket protocol in **RFC 6455**.

It's primarily intended at maintainers. It may also be useful for users who wish to understand what happens under the hood.

> **Warning:** Internals described in this document may change at any time.
>
> Backwards compatibility is only guaranteed for public APIs.

4.1.1 Lifecycle

State

WebSocket connections go through a trivial state machine:

- `CONNECTING`: initial state,
- `OPEN`: when the opening handshake is complete,
- `CLOSING`: when the closing handshake is started,
- `CLOSED`: when the TCP connection is closed.

Transitions happen in the following places:

- CONNECTING -> OPEN: in connection_open() which runs when the *opening handshake* completes and the WebSocket connection is established — not to be confused with connection_made() which runs when the TCP connection is established;
- OPEN -> CLOSING: in write_frame() immediately before sending a close frame; since receiving a close frame triggers sending a close frame, this does the right thing regardless of which side started the *closing handshake*; also in fail_connection() which duplicates a few lines of code from write_close_frame() and write_frame();
- * -> CLOSED: in connection_lost() which is always called exactly once when the TCP connection is closed.

Coroutines

The following diagram shows which coroutines are running at each stage of the connection lifecycle on the client side.

The lifecycle is identical on the server side, except inversion of control makes the equivalent of *connect()* implicit.

Coroutines shown in green are called by the application. Multiple coroutines may interact with the WebSocket connection concurrently.

Coroutines shown in gray manage the connection. When the opening handshake succeeds, connection_open() starts two tasks:

- transfer_data_task runs transfer_data() which handles incoming data and lets *recv()* consume it. It may be canceled to terminate the connection. It never exits with an exception other than CancelledError. See *data transfer* below.
- keepalive_ping_task runs keepalive_ping() which sends Ping frames at regular intervals and ensures that corresponding Pong frames are received. It is canceled when the connection terminates. It never exits with an exception other than CancelledError.
- close_connection_task runs close_connection() which waits for the data transfer to terminate, then takes care of closing the TCP connection. It must not be canceled. It never exits with an exception. See *connection termination* below.

Besides, fail_connection() starts the same close_connection_task when the opening handshake fails, in order to close the TCP connection.

Splitting the responsibilities between two tasks makes it easier to guarantee that websockets can terminate connections:

- within a fixed timeout,
- without leaking pending tasks,
- without leaking open TCP connections,

regardless of whether the connection terminates normally or abnormally.

transfer_data_task completes when no more data will be received on the connection. Under normal circumstances, it exits after exchanging close frames.

close_connection_task completes when the TCP connection is closed.

4.1.2 Opening handshake

`websockets` performs the opening handshake when establishing a WebSocket connection. On the client side, *connect()* executes it before returning the protocol to the caller. On the server side, it's executed before passing the protocol to the `ws_handler` coroutine handling the connection.

While the opening handshake is asymmetrical — the client sends an HTTP Upgrade request and the server replies with an HTTP Switching Protocols response — `websockets` aims at keeping the implementation of both sides consistent with one another.

On the client side, *handshake()*:

- builds a HTTP request based on the `uri` and parameters passed to *connect()*;
- writes the HTTP request to the network;
- reads a HTTP response from the network;
- checks the HTTP response, validates `extensions` and `subprotocol`, and configures the protocol accordingly;
- moves to the `OPEN` state.

On the server side, *handshake()*:

- reads a HTTP request from the network;
- calls *process_request()* which may abort the WebSocket handshake and return a HTTP response instead; this hook only makes sense on the server side;
- checks the HTTP request, negotiates `extensions` and `subprotocol`, and configures the protocol accordingly;
- builds a HTTP response based on the above and parameters passed to *serve()*;
- writes the HTTP response to the network;
- moves to the `OPEN` state;
- returns the `path` part of the `uri`.

The most significant asymmetry between the two sides of the opening handshake lies in the negotiation of extensions and, to a lesser extent, of the subprotocol. The server knows everything about both sides and decides what the parameters should be for the connection. The client merely applies them.

If anything goes wrong during the opening handshake, `websockets` *fails the connection*.

4.1.3 Data transfer

Symmetry

Once the opening handshake has completed, the WebSocket protocol enters the data transfer phase. This part is almost symmetrical. There are only two differences between a server and a client:

- client-to-server masking: the client masks outgoing frames; the server unmasks incoming frames;
- closing the TCP connection: the server closes the connection immediately; the client waits for the server to do it.

These differences are so minor that all the logic for data framing, for sending and receiving data and for closing the connection is implemented in the same class, *WebSocketCommonProtocol*.

The is_client attribute tells which side a protocol instance is managing. This attribute is defined on the *WebSocketServerProtocol* and *WebSocketClientProtocol* classes.

Data flow

The following diagram shows how data flows between an application built on top of websockets and a remote endpoint. It applies regardless of which side is the server or the client.

Public methods are shown in green, private methods in yellow, and buffers in orange. Methods related to connection termination are omitted; connection termination is discussed in another section below.

Receiving data

The left side of the diagram shows how websockets receives data.

Incoming data is written to a StreamReader in order to implement flow control and provide backpressure on the TCP connection.

transfer_data_task, which is started when the WebSocket connection is established, processes this data.

When it receives data frames, it reassembles fragments and puts the resulting messages in the messages queue.

When it encounters a control frame:

- if it's a close frame, it starts the closing handshake;
- if it's a ping frame, it answers with a pong frame;
- if it's a pong frame, it acknowledges the corresponding ping (unless it's an unsolicited pong).

Running this process in a task guarantees that control frames are processed promptly. Without such a task, websockets would depend on the application to drive the connection by having exactly one coroutine awaiting *recv()* at any time. While this happens naturally in many use cases, it cannot be relied upon.

Then *recv()* fetches the next message from the messages queue, with some complexity added for handling backpressure and termination correctly.

Sending data

The right side of the diagram shows how websockets sends data.

send() writes one or several data frames containing the message. While sending a fragmented message, concurrent calls to *send()* are put on hold until all fragments are sent. This makes concurrent calls safe.

ping() writes a ping frame and yields a Future which will be completed when a matching pong frame is received.

pong() writes a pong frame.

close() writes a close frame and waits for the TCP connection to terminate.

Outgoing data is written to a StreamWriter in order to implement flow control and provide backpressure from the TCP connection.

Closing handshake

When the other side of the connection initiates the closing handshake, `read_message()` receives a close frame while in the `OPEN` state. It moves to the `CLOSING` state, sends a close frame, and returns `None`, causing `transfer_data_task` to terminate.

When this side of the connection initiates the closing handshake with `close()`, it moves to the `CLOSING` state and sends a close frame. When the other side sends a close frame, `read_message()` receives it in the `CLOSING` state and returns `None`, also causing `transfer_data_task` to terminate.

If the other side doesn't send a close frame within the connection's close timeout, `websockets` *fails the connection*.

The closing handshake can take up to `2 * close_timeout`: one `close_timeout` to write a close frame and one `close_timeout` to receive a close frame.

Then `websockets` terminates the TCP connection.

4.1.4 Connection termination

`close_connection_task`, which is started when the WebSocket connection is established, is responsible for eventually closing the TCP connection.

First `close_connection_task` waits for `transfer_data_task` to terminate, which may happen as a result of:

- a successful closing handshake: as explained above, this exits the infinite loop in `transfer_data_task`;
- a timeout while waiting for the closing handshake to complete: this cancels `transfer_data_task`;
- a protocol error, including connection errors: depending on the exception, `transfer_data_task` *fails the connection* with a suitable code and exits.

`close_connection_task` is separate from `transfer_data_task` to make it easier to implement the timeout on the closing handshake. Canceling `transfer_data_task` creates no risk of canceling `close_connection_task` and failing to close the TCP connection, thus leaking resources.

Then `close_connection_task` cancels `keepalive_ping`. This task has no protocol compliance responsibilities. Terminating it to avoid leaking it is the only concern.

Terminating the TCP connection can take up to `2 * close_timeout` on the server side and `3 * close_timeout` on the client side. Clients start by waiting for the server to close the connection, hence the extra `close_timeout`. Then both sides go through the following steps until the TCP connection is lost: half-closing the connection (only for non-TLS connections), closing the connection, aborting the connection. At this point the connection drops regardless of what happens on the network.

4.1.5 Connection failure

If the opening handshake doesn't complete successfully, `websockets` fails the connection by closing the TCP connection.

Once the opening handshake has completed, `websockets` fails the connection by canceling `transfer_data_task` and sending a close frame if appropriate.

`transfer_data_task` exits, unblocking `close_connection_task`, which closes the TCP connection.

4.1.6 Server shutdown

WebSocketServer closes asynchronously like `asyncio.Server`. The shutdown happen in two steps:

1. Stop listening and accepting new connections;
2. Close established connections with close code 1001 (going away) or, if the opening handshake is still in progress, with HTTP status code 503 (Service Unavailable).

The first call to *close* starts a task that performs this sequence. Further calls are ignored. This is the easiest way to make *close* and *wait_closed* idempotent.

4.1.7 Cancellation

User code

`websockets` provides a WebSocket application server. It manages connections and passes them to user-provided connection handlers. This is an *inversion of control* scenario: library code calls user code.

If a connection drops, the corresponding handler should terminate. If the server shuts down, all connection handlers must terminate. Canceling connection handlers would terminate them.

However, using cancellation for this purpose would require all connection handlers to handle it properly. For example, if a connection handler starts some tasks, it should catch `CancelledError`, terminate or cancel these tasks, and then re-raise the exception.

Cancellation is tricky in `asyncio` applications, especially when it interacts with finalization logic. In the example above, what if a handler gets interrupted with `CancelledError` while it's finalizing the tasks it started, after detecting that the connection dropped?

`websockets` considers that cancellation may only be triggered by the caller of a coroutine when it doesn't care about the results of that coroutine anymore. (Source: Guido van Rossum). Since connection handlers run arbitrary user code, `websockets` has no way of deciding whether that code is still doing something worth caring about.

For these reasons, `websockets` never cancels connection handlers. Instead it expects them to detect when the connection is closed, execute finalization logic if needed, and exit.

Conversely, cancellation isn't a concern for WebSocket clients because they don't involve inversion of control.

Library

Most *public APIs* of `websockets` are coroutines. They may be canceled, for example if the user starts a task that calls these coroutines and cancels the task later. `websockets` must handle this situation.

Cancellation during the opening handshake is handled like any other exception: the TCP connection is closed and the exception is re-raised. This can only happen on the client side. On the server side, the opening handshake is managed by `websockets` and nothing results in a cancellation.

Once the WebSocket connection is established, internal tasks `transfer_data_task` and `close_connection_task` mustn't get accidentally canceled if a coroutine that awaits them is canceled. In other words, they must be shielded from cancellation.

recv() waits for the next message in the queue or for `transfer_data_task` to terminate, whichever comes first. It relies on `wait()` for waiting on two futures in parallel. As a consequence, even though it's waiting on a `Future` signaling the next message and on `transfer_data_task`, it doesn't propagate cancellation to them.

`ensure_open()` is called by *send()*, *ping()*, and *pong()*. When the connection state is `CLOSING`, it waits for `transfer_data_task` but shields it to prevent cancellation.

`close()` waits for the data transfer task to terminate with `wait_for()`. If it's canceled or if the timeout elapses, `transfer_data_task` is canceled, which is correct at this point. `close()` then waits for `close_connection_task` but shields it to prevent cancellation.

`close()` and `fail_connection()` are the only places where `transfer_data_task` may be canceled.

`close_connnection_task` starts by waiting for `transfer_data_task`. It catches `CancelledError` to prevent a cancellation of `transfer_data_task` from propagating to `close_connnection_task`.

4.1.8 Backpressure

Note: This section discusses backpressure from the perspective of a server but the concept applies to clients symmetrically.

With a naive implementation, if a server receives inputs faster than it can process them, or if it generates outputs faster than it can send them, data accumulates in buffers, eventually causing the server to run out of memory and crash.

The solution to this problem is backpressure. Any part of the server that receives inputs faster than it can process them and send the outputs must propagate that information back to the previous part in the chain.

`websockets` is designed to make it easy to get backpressure right.

For incoming data, `websockets` builds upon `StreamReader` which propagates backpressure to its own buffer and to the TCP stream. Frames are parsed from the input stream and added to a bounded queue. If the queue fills up, parsing halts until the application reads a frame.

For outgoing data, `websockets` builds upon `StreamWriter` which implements flow control. If the output buffers grow too large, it waits until they're drained. That's why all APIs that write frames are asynchronous.

Of course, it's still possible for an application to create its own unbounded buffers and break the backpressure. Be careful with queues.

4.1.9 Buffers

Note: This section discusses buffers from the perspective of a server but it applies to clients as well.

An asynchronous systems works best when its buffers are almost always empty.

For example, if a client sends data too fast for a server, the queue of incoming messages will be constantly full. The server will always be 32 messages (by default) behind the client. This consumes memory and increases latency for no good reason. The problem is called bufferbloat.

If buffers are almost always full and that problem cannot be solved by adding capacity — typically because the system is bottlenecked by the output and constantly regulated by backpressure — reducing the size of buffers minimizes negative consequences.

By default `websockets` has rather high limits. You can decrease them according to your application's characteristics.

Bufferbloat can happen at every level in the stack where there is a buffer. For each connection, the receiving side contains these buffers:

- OS buffers: tuning them is an advanced optimization.
- `StreamReader` bytes buffer: the default limit is 64 KiB. You can set another limit by passing a `read_limit` keyword argument to `connect()` or `serve()`.

- Incoming messages deque: its size depends both on the size and the number of messages it contains. By default the maximum UTF-8 encoded size is 1 MiB and the maximum number is 32. In the worst case, after UTF-8 decoding, a single message could take up to 4 MiB of memory and the overall memory consumption could reach 128 MiB. You should adjust these limits by setting the max_size and max_queue keyword arguments of *connect()* or *serve()* according to your application's requirements.

For each connection, the sending side contains these buffers:

- StreamWriter bytes buffer: the default size is 64 KiB. You can set another limit by passing a write_limit keyword argument to *connect()* or *serve()*.
- OS buffers: tuning them is an advanced optimization.

4.1.10 Concurrency

Awaiting any combination of *recv()*, *send()*, *close()* *ping()*, or *pong()* concurrently is safe, including multiple calls to the same method, with one exception and one limitation.

- **Only one coroutine can receive messages at a time.** This constraint avoids non-deterministic behavior (and simplifies the implementation). If a coroutine is awaiting *recv()*, awaiting it again in another coroutine raises RuntimeError.
- **Sending a fragmented message forces serialization.** Indeed, the WebSocket protocol doesn't support multiplexing messages. If a coroutine is awaiting *send()* to send a fragmented message, awaiting it again in another coroutine waits until the first call completes. This will be transparent in many cases. It may be a concern if the fragmented message is generated slowly by an asynchronous iterator.

Receiving frames is independent from sending frames. This isolates *recv()*, which receives frames, from the other methods, which send frames.

While the connection is open, each frame is sent with a single write. Combined with the concurrency model of asyncio, this enforces serialization. The only other requirement is to prevent interleaving other data frames in the middle of a fragmented message.

After the connection is closed, sending a frame raises *ConnectionClosed*, which is safe.

4.2 Limitations

The client doesn't attempt to guarantee that there is no more than one connection to a given IP address in a CONNECTING state.

The client doesn't support connecting through a proxy.

There is no way to fragment outgoing messages. A message is always sent in a single frame.

4.3 Security

4.3.1 Encryption

For production use, a server should require encrypted connections.

See this example of *encrypting connections with TLS*.

4.3.2 Memory use

> **Warning:** An attacker who can open an arbitrary number of connections will be able to perform a denial of service by memory exhaustion. If you're concerned by denial of service attacks, you must reject suspicious connections before they reach `websockets`, typically in a reverse proxy.

With the default settings, opening a connection uses 325 KiB of memory.

Sending some highly compressed messages could use up to 128 MiB of memory with an amplification factor of 1000 between network traffic and memory use.

Configuring a server to *optimize memory usage* will improve security in addition to improving performance.

4.3.3 Other limits

`websockets` implements additional limits on the amount of data it accepts in order to minimize exposure to security vulnerabilities.

In the opening handshake, `websockets` limits the number of HTTP headers to 256 and the size of an individual header to 4096 bytes. These limits are 10 to 20 times larger than what's expected in standard use cases. They're hard-coded. If you need to change them, monkey-patch the constants in `websockets.http`.

CHAPTER 5

Project

This is about websockets-the-project rather than websockets-the-software.

5.1 Contributing

Thanks for taking the time to contribute to websockets!

5.1.1 Code of Conduct

This project and everyone participating in it is governed by the Code of Conduct. By participating, you are expected to uphold this code. Please report inappropriate behavior to aymeric DOT augustin AT fractalideas DOT com.

(If I'm the person with the inappropriate behavior, please accept my apologies. I know I can mess up. I can't expect you to tell me, but if you choose to do so, I'll do my best to handle criticism constructively. – Aymeric)

5.1.2 Contributions

Bug reports, patches and suggestions are welcome!

Please open an issue or send a pull request.

Feedback about the documentation is especially valuable — the authors of `websockets` feel more confident about writing code than writing docs :-)

If you're wondering why things are done in a certain way, the *design document* provides lots of details about the internals of websockets.

5.1.3 Questions

GitHub issues aren't a good medium for handling questions. There are better places to ask questions, for example Stack Overflow.

If you want to ask a question anyway, please make sure that:

- it's a question about `websockets` and not about `asyncio`;
- it isn't answered by the documentation;
- it wasn't asked already.

A good question can be written as a suggestion to improve the documentation.

5.1.4 Bitcoin users

websockets appears to be quite popular for interfacing with Bitcoin or other cryptocurrency trackers. I'm strongly opposed to Bitcoin's carbon footprint.

Please stop heating the planet where my children are supposed to live, thanks.

Since `websockets` is released under an open-source license, you can use it for any purpose you like. However, I won't spend any of my time to help.

I will summarily close issues related to Bitcoin or cryptocurrency in any way.

5.2 Changelog

5.2.1 8.1

In development

5.2.2 8.0.1

- Restored the ability to import `WebSocketProtocolError` from `websockets`.

5.2.3 8.0

Warning: Version 8.0 drops compatibility with Python 3.4 and 3.5.

Note: Version 8.0 expects `process_request` to be a coroutine.

Previously, it could be a function or a coroutine.

If you're passing a `process_request` argument to *`serve()`* or *`WebSocketServerProtocol`*, or if you're overriding `process_request()` in a subclass, define it with `async def` instead of `def`.

For backwards compatibility, functions are still mostly supported, but mixing functions and coroutines won't work in some inheritance scenarios.

Note: Version 8.0 changes the behavior of the `max_queue` parameter.

If you were setting `max_queue=0` to make the queue of incoming messages unbounded, change it to `max_queue=None`.

Note: Version 8.0 deprecates the `host` **,** `port` **, and** `secure` **attributes of** *WebSocketCommonProtocol*.

Use *local_address* in servers and *remote_address* in clients instead of `host` and `port`.

Note: Version 8.0 renames the `WebSocketProtocolError` **exception** to `ProtocolError` **.**

A `WebSocketProtocolError` alias provides backwards compatibility.

Note: Version 8.0 adds the reason phrase to the return type of the low-level API *read_response()* **.**

Also:

- *send()*, *ping()*, and *pong()* support bytes-like types `bytearray` and `memoryview` in addition to `bytes`.
- Added *ConnectionClosedOK* and *ConnectionClosedError* subclasses of *ConnectionClosed* to tell apart normal connection termination from errors.
- Added *basic_auth_protocol_factory()* to enforce HTTP Basic Auth on the server side.
- *connect()* handles redirects from the server during the handshake.
- *connect()* supports overriding `host` and `port`.
- Added *unix_connect()* for connecting to Unix sockets.
- Improved support for sending fragmented messages by accepting asynchronous iterators in *send()*.
- Prevented spurious log messages about *ConnectionClosed* exceptions in keepalive ping task. If you were using `ping_timeout=None` as a workaround, you can remove it.
- Changed *WebSocketServer.close()* to perform a proper closing handshake instead of failing the connection.
- Avoided a crash when a `extra_headers` callable returns `None`.
- Improved error messages when HTTP parsing fails.
- Enabled readline in the interactive client.
- Added type hints (**PEP 484**).
- Added a FAQ to the documentation.
- Added documentation for extensions.
- Documented how to optimize memory usage.
- Improved API documentation.

5.2.4 7.0

Warning: Version 7.0 renames the `timeout` **argument of** *serve()* **and** *connect()* **to** `close_timeout` **.**

This prevents confusion with `ping_timeout`.

For backwards compatibility, `timeout` is still supported.

Warning: Version 7.0 changes how a server terminates connections when it's closed with *close()*.

Previously, connections handlers were canceled. Now, connections are closed with close code 1001 (going away). From the perspective of the connection handler, this is the same as if the remote endpoint was disconnecting. This removes the need to prepare for `CancelledError` in connection handlers.

You can restore the previous behavior by adding the following line at the beginning of connection handlers:

```
def handler(websocket, path):
    closed = asyncio.ensure_future(websocket.wait_closed())
    closed.add_done_callback(lambda task: task.cancel())
```

Note: Version 7.0 changes how a *ping()* that hasn't received a pong yet behaves when the connection is closed.

The ping — as in `ping = await websocket.ping()` — used to be canceled when the connection is closed, so that `await ping` raised `CancelledError`. Now `await ping` raises *ConnectionClosed* like other public APIs.

Note: Version 7.0 raises a `RuntimeError` exception if two coroutines call *recv()* concurrently.

Concurrent calls lead to non-deterministic behavior because there are no guarantees about which coroutine will receive which message.

Also:

- `websockets` sends Ping frames at regular intervals and closes the connection if it doesn't receive a matching Pong frame. See *WebSocketCommonProtocol* for details.
- Added `process_request` and `select_subprotocol` arguments to *serve()* and *WebSocketServerProtocol* to customize *process_request()* and *select_subprotocol()* without subclassing *WebSocketServerProtocol*.
- Added support for sending fragmented messages.
- Added the *wait_closed()* method to protocols.
- Added an interactive client: python -m websockets <uri>.
- Changed the `origins` argument to represent the lack of an origin with `None` rather than `''`.
- Fixed a data loss bug in *recv()*: canceling it at the wrong time could result in messages being dropped.
- Improved handling of multiple HTTP headers with the same name.
- Improved error messages when a required HTTP header is missing.

5.2.5 6.0

Warning: Version 6.0 introduces the *Headers* class for managing HTTP headers and changes several public APIs:

- *process_request()* now receives a *Headers* instead of a `HTTPMessage` in the `request_headers` argument.
- The `request_headers` and `response_headers` attributes of *WebSocketCommonProtocol* are *Headers* instead of `HTTPMessage`.
- The `raw_request_headers` and `raw_response_headers` attributes of *WebSocketCommonProtocol* are removed. Use *raw_items()* instead.
- Functions defined in the `handshake` module now receive *Headers* in argument instead of `get_header` or `set_header` functions. This affects libraries that rely on low-level APIs.
- Functions defined in the `http` module now return HTTP headers as *Headers* instead of lists of `(name, value)` pairs.

Since *Headers* and `HTTPMessage` provide similar APIs, this change won't affect most of the code dealing with HTTP headers.

Also:

- Added compatibility with Python 3.7.

5.2.6 5.0.1

- Fixed a regression in the 5.0 release that broke some invocations of *serve()* and *connect()*.

5.2.7 5.0

Note: Version 5.0 fixes a security issue introduced in version 4.0.

Version 4.0 was vulnerable to denial of service by memory exhaustion because it didn't enforce `max_size` when decompressing compressed messages (CVE-2018-1000518).

Note: Version 5.0 adds a `user_info` field to the return value of *parse_uri()* and *WebSocketURI*.

If you're unpacking `WebSocketURI` into four variables, adjust your code to account for that fifth field.

Also:

- *connect()* performs HTTP Basic Auth when the URI contains credentials.
- Iterating on incoming messages no longer raises an exception when the connection terminates with close code 1001 (going away).
- A plain HTTP request now receives a 426 Upgrade Required response and doesn't log a stack trace.
- *unix_serve()* can be used as an asynchronous context manager on Python 3.5.1.
- Added the *closed* property to protocols.

- If a `ping()` doesn't receive a pong, it's canceled when the connection is closed.
- Reported the cause of `ConnectionClosed` exceptions.
- Added new examples in the documentation.
- Updated documentation with new features from Python 3.6.
- Improved several other sections of the documentation.
- Fixed missing close code, which caused `TypeError` on connection close.
- Fixed a race condition in the closing handshake that raised `InvalidState`.
- Stopped logging stack traces when the TCP connection dies prematurely.
- Prevented writing to a closing TCP connection during unclean shutdowns.
- Made connection termination more robust to network congestion.
- Prevented processing of incoming frames after failing the connection.

5.2.8 4.0.1

- Fixed issues with the packaging of the 4.0 release.

5.2.9 4.0

Warning: Version 4.0 enables compression with the permessage-deflate extension.

In August 2017, Firefox and Chrome support it, but not Safari and IE.

Compression should improve performance but it increases RAM and CPU use.

If you want to disable compression, add `compression=None` when calling `serve()` or `connect()`.

Warning: Version 4.0 drops compatibility with Python 3.3.

Note: Version 4.0 removes the `state_name` attribute of protocols.

Use `protocol.state.name` instead of `protocol.state_name`.

Also:

- `WebSocketCommonProtocol` instances can be used as asynchronous iterators on Python 3.6. They yield incoming messages.
- Added `unix_serve()` for listening on Unix sockets.
- Added the `sockets` attribute to the return value of `serve()`.
- Reorganized and extended documentation.
- Aborted connections if they don't close within the configured `timeout`.
- Rewrote connection termination to increase robustness in edge cases.

- Stopped leaking pending tasks when `cancel()` is called on a connection while it's being closed.
- Reduced verbosity of "Failing the WebSocket connection" logs.
- Allowed `extra_headers` to override `Server` and `User-Agent` headers.

5.2.10 3.4

- Renamed `serve()` and `connect()`'s `klass` argument to `create_protocol` to reflect that it can also be a callable. For backwards compatibility, `klass` is still supported.
- `serve()` can be used as an asynchronous context manager on Python 3.5.1.
- Added support for customizing handling of incoming connections with `process_request()`.
- Made read and write buffer sizes configurable.
- Rewrote HTTP handling for simplicity and performance.
- Added an optional C extension to speed up low-level operations.
- An invalid response status code during `connect()` now raises `InvalidStatusCode` with a `code` attribute.
- Providing a `sock` argument to `connect()` no longer crashes.

5.2.11 3.3

- Ensured compatibility with Python 3.6.
- Reduced noise in logs caused by connection resets.
- Avoided crashing on concurrent writes on slow connections.

5.2.12 3.2

- Added `timeout`, `max_size`, and `max_queue` arguments to `connect()` and `serve()`.
- Made server shutdown more robust.

5.2.13 3.1

- Avoided a warning when closing a connection before the opening handshake.
- Added flow control for incoming data.

5.2.14 3.0

Warning: Version 3.0 introduces a backwards-incompatible change in the `recv()` API.

If you're upgrading from 2.x or earlier, please read this carefully.

`recv()` used to return `None` when the connection was closed. This required checking the return value of every call:

```
message = await websocket.recv()
if message is None:
    return
```

Now it raises a `ConnectionClosed` exception instead. This is more Pythonic. The previous code can be simplified to:

```
message = await websocket.recv()
```

When implementing a server, which is the more popular use case, there's no strong reason to handle such exceptions. Let them bubble up, terminate the handler coroutine, and the server will simply ignore them.

In order to avoid stranding projects built upon an earlier version, the previous behavior can be restored by passing `legacy_recv=True` to `serve()`, `connect()`, `WebSocketServerProtocol`, or `WebSocketClientProtocol`. `legacy_recv` isn't documented in their signatures but isn't scheduled for deprecation either.

Also:

- `connect()` can be used as an asynchronous context manager on Python 3.5.1.
- Updated documentation with `await` and `async` syntax from Python 3.5.
- `ping()` and `pong()` support data passed as `str` in addition to `bytes`.
- Worked around an asyncio bug affecting connection termination under load.
- Made `state_name` attribute on protocols a public API.
- Improved documentation.

5.2.15 2.7

- Added compatibility with Python 3.5.
- Refreshed documentation.

5.2.16 2.6

- Added `local_address` and `remote_address` attributes on protocols.
- Closed open connections with code 1001 when a server shuts down.
- Avoided TCP fragmentation of small frames.

5.2.17 2.5

- Improved documentation.
- Provided access to handshake request and response HTTP headers.
- Allowed customizing handshake request and response HTTP headers.
- Supported running on a non-default event loop.
- Returned a 403 status code instead of 400 when the request Origin isn't allowed.
- Canceling `recv()` no longer drops the next message.
- Clarified that the closing handshake can be initiated by the client.
- Set the close code and reason more consistently.
- Strengthened connection termination by simplifying the implementation.

- Improved tests, added tox configuration, and enforced 100% branch coverage.

5.2.18 2.4

- Added support for subprotocols.
- Supported non-default event loop.
- Added `loop` argument to *connect()* and *serve()*.

5.2.19 2.3

- Improved compliance of close codes.

5.2.20 2.2

- Added support for limiting message size.

5.2.21 2.1

- Added `host`, `port` and `secure` attributes on protocols.
- Added support for providing and checking Origin.

5.2.22 2.0

Warning: Version 2.0 introduces a backwards-incompatible change in the *send()*, *ping()*, **and** *pong()* **APIs.**

If you're upgrading from 1.x or earlier, please read this carefully.

These APIs used to be functions. Now they're coroutines.

Instead of:

```
websocket.send(message)
```

you must now write:

```
await websocket.send(message)
```

Also:

- Added flow control for outgoing data.

5.2.23 1.0

- Initial public release.

5.3 License

```
Copyright (c) 2013-2019 Aymeric Augustin and contributors.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright notice,
      this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright notice,
      this list of conditions and the following disclaimer in the documentation
      and/or other materials provided with the distribution.
    * Neither the name of websockets nor the names of its contributors may
      be used to endorse or promote products derived from this software without
      specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Python Module Index

W

Index

L

M

N

O

P

R

S

U

W