websockets Documentation

Release 5.0

Aymeric Augustin

May 24, 2018

Contents

1 Tutorials **3**
- 1.1 Getting started . . . 3

2 How-to guides **15**
- 2.1 Cheat sheet . . . 15
- 2.2 Deployment . . . 17

3 Reference **21**
- 3.1 API . . . 21

4 Discussions **35**
- 4.1 Design . . . 35
- 4.2 Limitations . . . 41
- 4.3 Security . . . 41

5 Project **43**
- 5.1 Contributing . . . 43
- 5.2 Changelog . . . 43
- 5.3 License . . . 48

Python Module Index **49**

`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 a client that says "Hello world!":

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

import asyncio
import websockets

async def hello(uri):
    async with websockets.connect(uri) as websocket:
        await websocket.send("Hello world!")

asyncio.get_event_loop().run_until_complete(
    hello('ws://localhost:8765'))
```

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)

asyncio.get_event_loop().run_until_complete(
    websockets.serve(echo, 'localhost', 8765))
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 3.4.

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.

For the best experience, you should start with Python 3.6. `asyncio` received interesting improvements between Python 3.4 and 3.6.

Warning: This documentation is written for Python 3.6. If you're using an older Python version, you need to *adapt the code samples*.

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.

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

# WS server example

import asyncio
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}")

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():
    async with websockets.connect(
            'ws://localhost:8765') 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 TLS. WSS requires TLS certificates like HTTPS.

Here's how to adapt the server example to provide secure connections, using APIs available in Python 3.6.

Refer to the documentation of the `ssl` module for configuring the context securely or adapting the code to older Python versions.

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

# 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)
ssl_context.load_cert_chain(
    pathlib.Path(__file__).with_name('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, also on Python 3.6.

```
#!/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)
ssl_context.load_verify_locations(
    pathlib.Path(__file__).with_name('localhost.pem'))

async def hello():
    async with websockets.connect(
            'wss://localhost:8765', 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):
    # 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':
```

(continues on next page)

(continued from previous page)

```
                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)

asyncio.get_event_loop().run_until_complete(
    websockets.serve(counter, 'localhost', 6789))
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;
            }
            .value {
                min-width: 2em;
            }
            .state {
                font-size: 2em;
            }
        </style>
    </head>
```

(continues on next page)

(continued from previous page)

```
    <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.

Asynchronous iteration was introduced in Python 3.6; here's the same code for earlier Python versions:

```
async def consumer_handler(websocket, path):
    while True:
        message = await websocket.recv()
        await consumer(message)
```

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

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):
```

(continues on next page)

(continued from previous page)

```
    # Register.
    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 Python < 3.6

This documentation takes advantage of several features that aren't available in Python < 3.6:

- `await` and `async` were added in Python 3.5;
- Asynchronous context managers didn't work well until Python 3.5.1;
- Asynchronous iterators were added in Python 3.6;
- f-strings were introduced in Python 3.6 (this is unrelated to `asyncio` and `websockets`).

Here's how to adapt the basic server example.

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

# WS server example for old Python versions

import asyncio
import websockets

@asyncio.coroutine
def hello(websocket, path):
    name = yield from websocket.recv()
    print("< {}".format(name))

    greeting = "Hello {}!".format(name)

    yield from websocket.send(greeting)
    print("> {}".format(greeting))

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

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

And here's the basic client example.

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

# WS client example for old Python versions

import asyncio
import websockets

@asyncio.coroutine
def hello():
    websocket = yield from websockets.connect(
        'ws://localhost:8765/')

    try:
        name = input("What's your name? ")

        yield from websocket.send(name)
        print("> {}".format(name))

        greeting = yield from websocket.recv()
        print("< {}".format(greeting))

    finally:
        yield from websocket.close()

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

`await` and `async`

If you're using Python < 3.5, you must substitute:

```
async def ...
```

with:

```
@asyncio.coroutine
def ...
```

and:

```
await ...
```

with:

```
yield from ...
```

Otherwise you will encounter a `SyntaxError`.

Asynchronous context managers

Asynchronous context managers were added in Python 3.5. However, `websockets` only supports them on Python 3.5.1, where `ensure_future()` accepts any awaitable.

If you're using Python < 3.5.1, instead of:

```
with websockets.connect(...) as client:
    ...
```

you must write:

```
client = yield from websockets.connect(...)
try:
    ...
finally:
    yield from client.close()
```

Asynchronous iterators

If you're using Python < 3.6, you must replace:

```
async for message in websocket:
    ...
```

with:

```
while True:
    message = yield from websocket.recv()
    ...
```

The latter will always raise a *ConnectionClosed* exception when the connection is closed, while the former will only raise that exception if the connection terminates with an error.

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.
  - 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()`.
  - On Python 3.5.1, 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()`.
  - On Python 3.5.1, 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 Keeping connections open

Pinging the other side once in a while is a good way to check whether the connection is still working, and also to keep it open in case something kills idle connections after some time:

```
while True:
    try:
        msg = await asyncio.wait_for(ws.recv(), timeout=20)
    except asyncio.TimeoutError:
        # No data in 20 seconds, check the connection.
        try:
            pong_waiter = await ws.ping()
            await asyncio.wait_for(pong_waiter, timeout=10)
        except asyncio.TimeoutError:
            # No response to ping in 10 seconds, disconnect.
            break
    else:
        # do something with msg
        ...
```

2.1.5 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):
```

(continues on next page)

(continued from previous page)

```
    ...

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` corountine 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.

Tasks that handle connections will be cancelled. For example, if the handler is awaiting *`recv()`*, that call will raise `CancelledError`.

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()
```

(continues on next page)

(continued from previous page)

```
# The stop condition is set when receiving SIGTERM.
stop = asyncio.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))
```

`async` and `await` were introduced in Python 3.5. websockets supports asynchronous context managers on Python 3.5.1. `async for` was introduced in Python 3.6. Here's the equivalent for older Python versions:

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

import asyncio
import signal
import websockets

async def echo(websocket, path):
    while True:
        try:
            msg = await websocket.recv()
        except websockets.ConnectionClosed:
            break
        else:
            await websocket.send(msg)

loop = asyncio.get_event_loop()

# Create the server.
start_server = websockets.serve(echo, 'localhost', 8765)
server = loop.run_until_complete(start_server)

# Run the server until receiving SIGTERM.
stop = asyncio.Future()
loop.add_signal_handler(signal.SIGTERM, stop.set_result, None)
loop.run_until_complete(stop)

# Shut down the server.
server.close()
loop.run_until_complete(server.wait_closed())
```

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 use

In order to avoid excessive memory use caused by buffer bloat, it is strongly recommended to *tune buffer sizes*.

Most importantly `max_size` should be lowered according to the expected size of messages. It is also suggested to lower `max_queue`, `read_limit` and `write_limit` if memory use is a concern.

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

class ServerProtocol(websockets.WebSocketServerProtocol):

    async def process_request(self, 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, create_protocol=ServerProtocol)

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

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 functions to build and validate the request and response headers.

The second phase is the core of the WebSocket protocol. `websockets` provides a standalone implementation on top of `asyncio` with a very 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

The *`websockets.server`* module defines a simple WebSocket server API.

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

Create, start, and return a *WebSocketServer*.

serve() returns an awaitable. Awaiting it yields an instance of *WebSocketServer* which provides *close()* and *wait_closed()* methods for terminating the server and cleaning up its resources.

On Python 3.5, *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. Internally, it creates and starts a `Server` object by calling `create_server()`. The *WebSocketServer* it returns keeps a reference to this object.

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 asyncio protocol that manages the connection. It should be a callable or class accepting the same arguments as *WebSocketServerProtocol* and returning a *WebSocketServerProtocol* instance. It defaults to *WebSocketServerProtocol*.

The behavior of the `timeout`, `max_size`, and `max_queue`, `read_limit`, and `write_limit` optional arguments is described in the documentation of *WebSocketCommonProtocol*.

serve() also accepts the following optional arguments:

- `origins` defines acceptable Origin HTTP headers — include `''` 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 — it can be a mapping, an iterable of (name, value) pairs, or a callable taking the request path and headers in arguments.
- `compression` is a shortcut to configure compression extensions; by default it enables the "permessage-deflate" extension; set it to `None` to disable compression

Whenever a client connects, the server accepts the connection, creates a *WebSocketServerProtocol*, performs the opening handshake, and delegates to the WebSocket handler. Once the handler completes, the server performs the closing handshake and closes the connection.

When a server is closed with *close()*, all running WebSocket handlers are cancelled. They may intercept `CancelledError` and perform cleanup actions before re-raising that exception. If a handler started new tasks, it should cancel them as well in that case.

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())
```

websockets.server.**unix_serve**(*ws_handler*, *path*, ***, *create_protocol=None*, *timeout=10*, *max_size=2 ** 20*, *max_queue=2 ** 5*, *read_limit=2 ** 16*, *write_limit=2 ** 16*, *loop=None*, *origins=None*, *extensions=None*, *subprotocols=None*, *extra_headers=None*, *compression='deflate'*, ***kwds*)

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.

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

Wrapper for `Server` that closes connections on exit.

This class provides the return type of *serve()*.

It mimics the interface of `AbstractServer`, namely its `close()` and `wait_closed()` methods, to close WebSocket connections properly on exit, in addition to closing the underlying `Server`.

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 underlying server, and clean up connections.

This calls `close()` on the underlying `Server` object, closes open connections with status code 1001, and stops accepting new connections.

wait_closed()

Wait until the underlying server and all connections are closed.

This calls `wait_closed()` on the underlying `Server` object and waits until closing handshakes are complete and all connections are closed.

This method must be called after *close()*.

sockets

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

`None` if the server is closed.

class websockets.server.**WebSocketServerProtocol**(*ws_handler*, *ws_server*, ***, *host=None*, *port=None*, *secure=None*, *timeout=10*, *max_size=2 ** 20*, *max_queue=2 ** 5*, *read_limit=2 ** 16*, *write_limit=2 ** 16*, *loop=None*, *origins=None*, *extensions=None*, *subprotocols=None*, *extra_headers=None*)

Complete WebSocket server implementation as an `asyncio.Protocol`.

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.

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

Perform the server side of the opening handshake.

If provided, `origins` is a list of acceptable HTTP Origin values. Include `''` if the lack of an origin is acceptable.

If provided, `available_extensions` is a list of supported extensions in the order in which they should be used.

If provided, `available_subprotocols` is a list of supported subprotocols in order of decreasing preference.

If provided, `extra_headers` sets additional HTTP response headers. It can be a mapping or an iterable of (name, value) pairs. It can also be a callable taking the request path and headers in arguments.

Raise *InvalidHandshake* if the handshake fails.

Return the path of the URI of the request.

process_request (*path*, *request_headers*)

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

`request_headers` are a `HTTPMessage`.

If this coroutine returns `None`, the WebSocket handshake continues. If it returns a status code, headers and a optionally a response body, that HTTP response is sent and the connection is closed.

The HTTP status must be a `HTTPStatus`. HTTP headers must be an iterable of `(name, value)` pairs. If provided, the HTTP response body must be `bytes`.

(`HTTPStatus` was added in Python 3.5. Use a compatible object on earlier versions. Look at `SWITCHING_PROTOCOLS` in `websockets.compatibility` for an example.)

This method may be overridden to check the request headers and set a different status, for example to authenticate the request and return `HTTPStatus.UNAUTHORIZED` or `HTTPStatus.FORBIDDEN`.

It is declared as a coroutine because such authentication checks are likely to require network requests.

static 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 subprotocols are 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.

Client

The *websockets.client* module defines a simple WebSocket client API.

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

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

connect() returns an awaitable. Awaiting it yields an instance of *WebSocketClientProtocol* which can then be used to send and receive messages.

On Python 3.5.1, *connect()* can 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.

The behavior of the `timeout`, `max_size`, and `max_queue`, `read_limit`, and `write_limit` optional arguments is described in the documentation of *WebSocketCommonProtocol*.

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

connect() also accepts the following optional arguments:

- `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 mapping or an iterable of (name, value) pairs
- `compression` is a shortcut to configure compression extensions; by default it enables the "permessage-deflate" extension; set it to `None` to disable compression

connect() raises `InvalidURI` if `uri` is invalid and `InvalidHandshake` if the opening handshake fails.

class websockets.client.**WebSocketClientProtocol**(**, host=None, port=None, secure=None, timeout=10, max_size=2 ** 20, max_queue=2 ** 5, read_limit=2 ** 16, write_limit=2 ** 16, loop=None, origin=None, extensions=None, subprotocols=None, extra_headers=None*)

Complete WebSocket client implementation as an `asyncio.Protocol`.

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

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

Perform the client side of the opening handshake.

If provided, `origin` sets the Origin HTTP header.

If provided, `available_extensions` is a list of supported extensions in the order in which they should be used.

If provided, `available_subprotocols` is a list of supported subprotocols in order of decreasing preference.

If provided, `extra_headers` sets additional HTTP request headers. It must be a mapping or an iterable of (name, value) pairs.

Raise *InvalidHandshake* if the handshake fails.

Shared

The *websockets.protocol* module handles WebSocket control and data frames as specified in sections 4 to 8 of RFC 6455.

class websockets.protocol.**WebSocketCommonProtocol**(*, *host=None*, *port=None*, *secure=None*, *timeout=10*, *max_size=2 ** 20*, *max_queue=2 ** 5*, *read_limit=2 ** 16*, *write_limit=2 ** 16*, *loop=None*)

This class implements common parts of the WebSocket protocol.

It assumes that the WebSocket connection is established. The handshake is managed in subclasses such as *WebSocketServerProtocol* and *WebSocketClientProtocol*.

It runs a task that stores incoming data frames in a queue and deals with control frames automatically. It sends outgoing data frames and performs the closing handshake.

On Python 3.6, *WebSocketCommonProtocol* instances support 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 status code 1000 (OK) or 1001 (going away). It raises a *ConnectionClosed* exception when the connection is closed with any other status code.

The `host`, `port` and `secure` parameters are simply stored as attributes for handlers that need them.

The `timeout` parameter defines the maximum wait time in seconds for completing the closing handshake and, only on the client side, for terminating the TCP connection. *close()* will complete in at most `4 * timeout` on the server side and `5 * timeout` on the client side.

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

The `max_queue` parameter sets the maximum length of the queue that holds incoming messages. The default value is 32. 0 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 websocket connection may use up to `4 * max_size * max_queue` bytes of memory to store incoming messages. By default, this is 128MB. 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 64kB, 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 64kB, 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, and the request and response HTTP headers are available:

- as a `HTTPMessage` in the `request_headers` and `response_headers` attributes
- as an iterable of (name, value) pairs in the `raw_request_headers` and `raw_response_headers` attributes

These attributes must be treated as immutable.

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

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

close(*code=1000*, *reason=''*)
This coroutine performs the closing handshake.

It waits for the other end to complete the handshake and for the TCP connection to terminate.

It doesn't do anything once the connection is closed. In other words it's idemptotent.

It's safe to wrap this coroutine in `ensure_future()` since errors during connection termination aren't particularly useful.

`code` must be an `int` and `reason` a `str`.

recv()
This coroutine receives the next message.

It returns a `str` for a text frame and `bytes` for a binary frame.

When the end of the message stream is reached, *`recv()`* raises *`ConnectionClosed`*. This can happen after a normal connection closure, a protocol error or a network failure.

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

send(*data*)
This coroutine sends a message.

It sends `str` as a text frame and `bytes` as a binary frame. It raises a `TypeError` for other inputs.

ping(*data=None*)
This coroutine sends a ping.

It returns 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. The content may be overridden with the optional `data` argument which must be of type `str` (which will be encoded to UTF-8) or `bytes`.

pong(*data=b''*)
This coroutine sends a pong.

An unsolicited pong may serve as a unidirectional heartbeat.

The content may be overridden with the optional `data` argument which must be of type `str` (which will be encoded to UTF-8) or `bytes`.

local_address
Local address of the connection.

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

remote_address
Remote address of the connection.

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

open
This property is `True` when the connection is usable.

It may be used to detect disconnections but this is discouraged per the EAFP principle. When `open` is `False`, using the connection raises a *`ConnectionClosed`* exception.

closed

This property is `True` once the connection is closed.

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

Exceptions

exception `websockets.exceptions.`**AbortHandshake**(*status*, *headers*, *body=None*)
Exception raised to abort a handshake and return a HTTP response.

exception `websockets.exceptions.`**ConnectionClosed**(*code*, *reason*)
Exception raised when trying to read or write on a closed connection.

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

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

exception `websockets.exceptions.`**InvalidHandshake**
Exception raised when a handshake request or response is invalid.

exception `websockets.exceptions.`**InvalidHeader**(*name*, *value*)
Exception raised when a HTTP header doesn’t have a valid format or value.

exception `websockets.exceptions.`**InvalidHeaderFormat**(*name*, *error*, *string*, *pos*)
Exception raised when a Sec-WebSocket-* HTTP header cannot be parsed.

exception `websockets.exceptions.`**InvalidHeaderValue**(*name*, *value*)
Exception raised when a Sec-WebSocket-* HTTP header has a wrong value.

exception `websockets.exceptions.`**InvalidMessage**
Exception raised when the HTTP message in a handshake request is malformed.

exception `websockets.exceptions.`**InvalidOrigin**(*origin*)
Exception raised when the Origin header in a request isn’t allowed.

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

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

exception `websockets.exceptions.`**InvalidState**
Exception raised when an operation is forbidden in the current state.

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

Provides the integer status code in its `status_code` attribute.

exception `websockets.exceptions.`**InvalidUpgrade**(*name*, *value*)
Exception raised when a Upgrade or Connection header isn’t correct.

exception `websockets.exceptions.`**InvalidURI**
Exception raised when an URI isn’t a valid websocket URI.

exception `websockets.exceptions.`**NegotiationError**
Exception raised when negociating an extension fails.

exception websockets.exceptions.**PayloadTooBig**
Exception raised when a frame's payload exceeds the maximum size.

exception websockets.exceptions.**WebSocketProtocolError**
Internal exception raised when the remote side breaks the protocol.

3.1.3 Low-level

Opening handshake

The *websockets.handshake* module deals with the WebSocket opening handshake according to section 4 of RFC 6455.

It provides functions to implement the handshake with any existing HTTP library. You must pass to these functions:

- A `set_header` function accepting a header name and a header value,
- A `get_header` function accepting a header name and returning the header value.

The inputs and outputs of `get_header` and `set_header` are `str` objects containing only ASCII characters.

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**(*set_header*)
Build a handshake request to send to the server.

Return the `key` which must be passed to *check_response()*.

websockets.handshake.**check_request**(*get_header*)
Check a handshake request received from the client.

If the handshake is valid, this function returns the `key` which must be passed to *build_response()*.

Otherwise it raises an *InvalidHandshake* exception and the server must return an error like 400 Bad Request.

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.

websockets.handshake.**build_response**(*set_header*, *key*)
Build a handshake response to send to the client.

`key` comes from *check_request()*.

websockets.handshake.**check_response**(*get_header*, *key*)
Check a handshake response received from the server.

`key` comes from *build_request()*.

If the handshake is valid, this function returns `None`.

Otherwise it raises an *InvalidHandshake* exception.

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.

Data transfer

The *websockets.framing* module implements data framing as specified in section 5 of RFC 6455.

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

class websockets.framing.**Frame**

WebSocket frame.

- `fin` is the FIN bit
- `rsv1` is the RSV1 bit
- `rsv2` is the RSV2 bit
- `rsv3` is the RSV3 bit
- `opcode` is the opcode
- `data` is the payload data

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

check()

Check that this frame contains acceptable values.

Raise *WebSocketProtocolError* if this frame contains incorrect values.

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

Read a WebSocket frame and return a *Frame* object.

`reader` is a coroutine taking an integer argument and reading exactly this number of bytes, unless the end of file is reached.

`mask` is a `bool` telling whether the frame should be masked i.e. whether the read happens on the server side.

If `max_size` is set and the payload exceeds this size in bytes, *PayloadTooBig* is raised.

If `extensions` is provided, it's a list of classes with an `decode()` method that transform the frame and return a new frame. They are applied in reverse order.

This function validates the frame before returning it and raises *WebSocketProtocolError* if it contains incorrect values.

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

Write a WebSocket frame.

`frame` is the *Frame* object to write.

`writer` is a function accepting bytes.

`mask` is a `bool` telling whether the frame should be masked i.e. whether the write happens on the client side.

If `extensions` is provided, it's a list of classes with an `encode()` method that transform the frame and return a new frame. They are applied in order.

This function validates the frame before sending it and raises *WebSocketProtocolError* if it contains incorrect values.

websockets.framing.**encode_data**(*data*)
Helper that converts str or bytes to bytes.

str are encoded with UTF-8.

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

Return (code, reason) when code is an int and reason a str.

Raise *WebSocketProtocolError* or UnicodeDecodeError if the data is invalid.

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

This is the reverse of *parse_close()*.

URI parser

The *websockets.uri* module implements parsing of WebSocket URIs according to section 3 of RFC 6455.

websockets.uri.**parse_uri**(*uri*)
This function parses and validates a WebSocket URI.

If the URI is valid, it returns a *WebSocketURI*.

Otherwise it raises an *InvalidURI* exception.

class websockets.uri.**WebSocketURI**
WebSocket URI.

- secure is the secure flag
- host is the lower-case host
- port if the integer port, it's always provided even if it's the default
- resource_name is the resource name, that is, the path and optional query
- user_info is an (username, password) tuple when the URI contains User Information, else None.

host
Alias for field number 1

port
Alias for field number 2

resource_name
Alias for field number 3

secure
Alias for field number 0

user_info
Alias for field number 4

Utilities

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

Its functions cannot be imported from `websockets`. They must be imported from *websockets.headers*.

`websockets.headers.`**`parse_connection`**(*string*)
Parse a `Connection` header.

Return a list of connection options.

Raise *InvalidHeaderFormat* on invalid inputs.

`websockets.headers.`**`parse_upgrade`**(*string*)
Parse an `Upgrade` header.

Return a list of connection options.

Raise *InvalidHeaderFormat* on invalid inputs.

`websockets.headers.`**`parse_extension_list`**(*string*)
Parse a `Sec-WebSocket-Extensions` header.

Return a value with the following format:

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

Parameter values are `None` when no value is provided.

Raise *InvalidHeaderFormat* on invalid inputs.

`websockets.headers.`**`build_extension_list`**(*extensions*)
Unparse a `Sec-WebSocket-Extensions` header.

This is the reverse of *parse_extension_list()*.

`websockets.headers.`**`parse_subprotocol_list`**(*string*)
Parse a `Sec-WebSocket-Protocol` header.

Raise *InvalidHeaderFormat* on invalid inputs.

`websockets.headers.`**`build_subprotocol_list`**(*protocols*)
Unparse a `Sec-WebSocket-Protocol` header.

This is the reverse of *parse_subprotocol_list()*.

The *websockets.http* module provides basic HTTP parsing and serialization. It is merely adequate for WebSocket handshake messages.

Its functions cannot be imported from `websockets`. They must be imported from *websockets.http*.

`websockets.http.`**`read_request`**(*stream*)
Read an HTTP/1.1 GET request from `stream`.

`stream` is an `StreamReader`.

Return `(path, headers)` where `path` is a `str` and `headers` is a list of `(name, value)` tuples.

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

Non-ASCII characters are represented with surrogate escapes.

Raise an exception if the request isn't well formatted.

Don'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.

`websockets.http.`**`read_response`**(*stream*)
Read an HTTP/1.1 response from `stream`.

`stream` is an `StreamReader`.

Return `(status_code, headers)` where `status_code` is a `int` and `headers` is a list of `(name, value)` tuples.

Non-ASCII characters are represented with surrogate escapes.

Raise an exception if the response isn't well formatted.

Don'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.

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.

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 cancelled to terminate the connection. It never exits with an exception other than `CancelledError`. See *data transfer* below.
- `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 cancelled. 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 keepping 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, negociates `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 assymetry between the two sides of the opening handshake lies in the negociation 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 anwsers 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 termination correctly.

Sending data

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

`send()` writes a single data frame containing the message. Fragmentation isn't supported at this time.

`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 timeout, `websockets` *fails the connection*.

The closing handshake can take up to `2 * timeout`: one `timeout` to write a close frame and one `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` **:ref:'fails the connection <connection-failure>'_** 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. Cancelling `transfer_data_task` creates no risk of cancelling `close_connection_task` and failing to close the TCP connection, thus leaking resources.

Terminating the TCP connection can take up to `2 * timeout` on the server side and `3 * timeout` on the client side. Clients start by waiting for the server to close the connection, hence the extra `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 cancelling `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 Cancellation

Most *public APIs* of `websockets` are coroutines. They may be cancelled. `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 or logged.

Once the WebSocket connection is established, `transfer_data_task` and `close_connection_task` mustn't get accidentally cancelled if a coroutine that awaits them is cancelled. 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 tasks in parallel. As a consequence, even though it's waiting on the transfer data task, it doesn't propagate cancellation to that task.

`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 cancelled or if the timout elapses, `transfer_data_task` is cancelled. `transfer_data_task` is expected to catch the cancellation and terminate properly. This is the only point where it may be cancelled.

`close()` then waits for `close_connection_task` but shields it to prevent cancellation.

`close_connnection_task` starts by waiting for `transfer_data_task`. Since `transfer_data_task` handles `CancelledError`, cancellation doesn't propagate to `close_connnection_task`.

4.1.7 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 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 some 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.

4.1.8 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 64kB. You can set another limit by passing a `read_limit` keyword argument to *`connect()`* or *`serve()`*.
- Incoming messages `Queue`: its size depends both on the size and the number of messages it contains. By default the maximum UTF-8 encoded size is 1MB and the maximum number is 32. In the worst case, after UTF-8 decoding, a single message could take up to 4MB of memory and the overall memory consumption could reach 128MB. 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 64kB. You can set another limit by passing a `write_limit` keyword argument to *`connect()`* or *`serve()`*.
- OS buffers: tuning them is an advanced optimization.

4.1.9 Concurrency

Calling any combination of *recv()*, *send()*, *close()* *ping()*, or *pong()* concurrently is safe, including multiple calls to the same method.

As shown above, receiving frames is independent from sending frames. That isolates *recv()*, which receives frames, from the other methods, which send frames.

While *recv()* supports being called multiple times concurrently, this is unlikely to be useful: when multiple callers are waiting for the next message, exactly one of them will get it, but there is no guarantee about which one.

Methods that send frames also support concurrent calls. While the connection is open, each frame is sent with a single write. Combined with the concurrency model of `asyncio`, this enforces serialization. After the connection is closed, sending a frame raises *ConnectionClosed*.

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 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.

The baseline memory use for a connection is about 20kB.

The incoming bytes buffer, incoming messages queue and outgoing bytes buffer contribute to the memory use of a connection. By default, each bytes buffer takes up to 64kB and the messages queue up to 128MB, which is very large.

Most applications use small messages. Setting `max_size` according to the application's requirements is strongly recommended. See *Buffers* for details about tuning buffers.

When compression is enabled, additional memory may be allocated for carrying the compression context across messages, depending on the context takeover and window size parameters. With the default configuration, this adds 320kB to the memory use for a connection.

You can reduce this amount by configuring the `PerMessageDeflate` extension with lower `server_max_window_bits` and `client_max_window_bits` values. These parameters default is 15. Lowering them to 11 is a good choice.

Finally, memory consumed by your application code also counts towards the memory use of a connection.

4.3.2 Other limits

`websockets` implements additional limits on the amount of data it accepts in order to mimimize 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 hardcoded. 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

Bug reports, patches and suggestions are welcome! Please open an issue or send a pull request.

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

5.2 Changelog

5.2.1 5.1

In development

5.2.2 5.0

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

websockets 4.0 was vulnerable to denial of service by memory exhaustion because it didn't enforce `max_size` when decompressing compressed messages.

Warning: 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 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 *closed()* property.
- If a *ping()* doesn't receive a pong, it's cancelled 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.

5.2.3 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 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.
- 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.

5.2.4 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.

5.2.5 3.3

- Reduced noise in logs caused by connection resets.
- Avoided crashing on concurrent writes on slow connections.

5.2.6 3.2

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

5.2.7 3.1

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

5.2.8 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` atttribute on protocols a public API.
- Improved documentation.

5.2.9 2.7

- Added compatibility with Python 3.5.
- Refreshed documentation.

5.2.10 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.

5.2.11 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.
- Cancelling *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.

5.2.12 2.4

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

5.2.13 2.3

- Improved compliance of close codes.

5.2.14 2.2

- Added support for limiting message size.

5.2.15 2.1

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

5.2.16 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.

5.2.17 1.0

- Initial public release.

5.3 License

Python Module Index

w

websockets.client, 24
websockets.exceptions, 28
websockets.framing, 30
websockets.handshake, 29
websockets.headers, 32
websockets.http, 32
websockets.protocol, 25
websockets.server, 21
websockets.uri, 31

Index

A

AbortHandshake, 28

B

build_extension_list() (in module websockets.headers), 32
build_request() (in module websockets.handshake), 29
build_response() (in module websockets.handshake), 29
build_subprotocol_list() (in module websockets.headers), 32

C

check() (websockets.framing.Frame method), 30
check_request() (in module websockets.handshake), 29
check_response() (in module websockets.handshake), 29
close() (websockets.protocol.WebSocketCommonProtocol method), 27
close() (websockets.server.WebSocketServer method), 23
closed (websockets.protocol.WebSocketCommonProtocol attribute), 28
connect() (in module websockets.client), 24
ConnectionClosed, 28

D

DuplicateParameter, 28

E

encode_data() (in module websockets.framing), 31

F

Frame (class in websockets.framing), 30

H

handshake() (websockets.client.WebSocketClientProtocol method), 25
handshake() (websockets.server.WebSocketServerProtocol method), 23
host (websockets.uri.WebSocketURI attribute), 31

I

InvalidHandshake, 28
InvalidHeader, 28
InvalidHeaderFormat, 28
InvalidHeaderValue, 28
InvalidMessage, 28
InvalidOrigin, 28
InvalidParameterName, 28
InvalidParameterValue, 28
InvalidState, 28
InvalidStatusCode, 28
InvalidUpgrade, 28
InvalidURI, 28

L

local_address (websockets.protocol.WebSocketCommonProtocol attribute), 27

N

NegotiationError, 28

O

open (websockets.protocol.WebSocketCommonProtocol attribute), 27

P

parse_close() (in module websockets.framing), 31
parse_connection() (in module websockets.headers), 32
parse_extension_list() (in module websockets.headers), 32
parse_subprotocol_list() (in module websockets.headers), 32
parse_upgrade() (in module websockets.headers), 32
parse_uri() (in module websockets.uri), 31
PayloadTooBig, 28
ping() (websockets.protocol.WebSocketCommonProtocol method), 27

pong() (websockets.protocol.WebSocketCommonProtocol method), 27
port (websockets.uri.WebSocketURI attribute), 31
process_request() (websockets.server.WebSocketServerProtocol method), 24

R

read() (websockets.framing.Frame class method), 30
read_request() (in module websockets.http), 32
read_response() (in module websockets.http), 33
recv() (websockets.protocol.WebSocketCommonProtocol method), 27
remote_address (websockets.protocol.WebSocketCommonProtocol attribute), 27
resource_name (websockets.uri.WebSocketURI attribute), 31
RFC
    RFC 6455, 35

S

secure (websockets.uri.WebSocketURI attribute), 31
select_subprotocol() (websockets.server.WebSocketServerProtocol static method), 24
send() (websockets.protocol.WebSocketCommonProtocol method), 27
serialize_close() (in module websockets.framing), 31
serve() (in module websockets.server), 21
sockets (websockets.server.WebSocketServer attribute), 23

U

unix_serve() (in module websockets.server), 22
user_info (websockets.uri.WebSocketURI attribute), 31

W

wait_closed() (websockets.server.WebSocketServer method), 23
WebSocketClientProtocol (class in websockets.client), 25
WebSocketCommonProtocol (class in websockets.protocol), 25
WebSocketProtocolError, 29
websockets.client (module), 24
websockets.exceptions (module), 28
websockets.framing (module), 30
websockets.handshake (module), 29
websockets.headers (module), 32
websockets.http (module), 32
websockets.protocol (module), 25
websockets.server (module), 21
websockets.uri (module), 31
WebSocketServer (class in websockets.server), 23
WebSocketServerProtocol (class in websockets.server), 23
WebSocketURI (class in websockets.uri), 31
write() (websockets.framing.Frame method), 30