websockets

Release 10.0

Aymeric Augustin

Oct 25, 2022

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

asyncio.run(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)

async def main():
    async with websockets.serve(echo, "localhost", 8765):
        await asyncio.Future()  # run forever

asyncio.run(main())
```

Do you like it? Let's dive in!

CHAPTER

ONE

GETTING STARTED

1.1 Requirements

websockets requires Python 3.7.

Use the most recent Python release

For each minor version (3.x), only the latest bugfix or security release (3.x.y) is officially supported.

1.2 Installation

Install websockets with:

```
pip install websockets
```

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}")
```

(continues on next page)

(continued from previous page)

```
async def main():
    async with websockets.serve(hello, "localhost", 8765):
        await asyncio.Future()  # run forever

asyncio.run(main())
```

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.run(hello())
```

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

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 (Transport Layer Security). `wss` requires certificates like `https`.

TLS vs. SSL

TLS is sometimes referred to as SSL (Secure Sockets Layer). SSL was an earlier encryption protocol; the name stuck.

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

# 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)

async def main():
    async with websockets.serve(hello, "localhost", 8765, ssl=ssl_context):
        await asyncio.Future()  # run forever

asyncio.run(main())
```

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}")
```

(continues on next page)

(continued from previous page)

```
asyncio.run(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.

1.5 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)

async def main():
    async with websockets.serve(time, "localhost", 5678):
        await asyncio.Future()  # run forever

asyncio.run(main())
```

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);
```

(continues on next page)

(continued from previous page)

```
            };
            document.body.appendChild(messages);
        </script>
    </body>
</html>
```

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 counter(websocket, path):
    try:
        # Register user
        USERS.add(websocket)
        websockets.broadcast(USERS, users_event())
        # Send current state to user
        await websocket.send(state_event())
        # Manage state changes
        async for message in websocket:
```

(continues on next page)

(continued from previous page)

```
            data = json.loads(message)
            if data["action"] == "minus":
                STATE["value"] -= 1
                websockets.broadcast(USERS, state_event())
            elif data["action"] == "plus":
                STATE["value"] += 1
                websockets.broadcast(USERS, state_event())
            else:
                logging.error("unsupported event: %s", data)
    finally:
        # Unregister user
        USERS.remove(websocket)
        websockets.broadcast(USERS, users_event())


async def main():
    async with websockets.serve(counter, "localhost", 6789):
        await asyncio.Future()  # run forever


if __name__ == "__main__":
    asyncio.run(main())
```

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 {
```

(continues on next page)

(continued from previous page)

```
            color: red;
        }
        .plus {
            color: green;
        }
        .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>
```

(continues on next page)

(continued from previous page)

```
</html>
```

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.

1.7.1 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.

1.7.2 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.

1.7.3 Both sides

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,
    )
```

(continues on next page)

(continued from previous page)

```
    for task in pending:
        task.cancel()
```

1.7.4 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.
    connected.add(websocket)
    try:
        # Broadcast a message to all connected clients.
        websockets.broadcast(connected, "Hello!")
        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.

1.8 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.

1.9 One more thing...

websockets provides an interactive client:

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

CHAPTER

TWO

HOW-TO GUIDES

If you're stuck, perhaps you'll find the answer in the FAQ or the cheat sheet.

2.1 FAQ

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

2.1.1 Server side

Why does my server close the connection prematurely?

Your connection handler exits prematurely. Wait for the work to be finished before returning.

For example, if your handler has a structure similar to:

```
async def handler(websocket, path):
    asyncio.create_task(do_some_work())
```

change it to:

```
async def handler(websocket, path):
    await do_some_work()
```

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 can I pass additional arguments to the connection handler?

You can bind additional arguments to the connection handler 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, ...)
```

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.

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"]
```

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?

You don't.

HTTP and WebSockets have widely different operational characteristics. Running them on the same server is a bad idea.

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.

2.1.2 Client side

Why does my client close the connection prematurely?

You're exiting the context manager prematurely. Wait for the work to be finished before exiting.

For example, if your code has a structure similar to:

```
async with connect(...) as websocket:
    asyncio.create_task(do_some_work())
```

change it to:

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

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 stop a client that is continuously processing messages?

You can close the connection.

Here's an example that terminates cleanly when it receives SIGTERM on Unix:

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

import asyncio
import signal
import websockets

async def client():
    uri = "ws://localhost:8765"
    async with websockets.connect(uri) as websocket:
        # Close the connection when receiving SIGTERM.
        loop = asyncio.get_running_loop()
        loop.add_signal_handler(
            signal.SIGTERM, loop.create_task, websocket.close())

        # Process messages received on the connection.
        async for message in websocket:
            ...

asyncio.run(client())
```

How do I disable TLS/SSL certificate verification?

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

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

2.1.3 asyncio usage

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.create_task()`.

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

Why does my program never receives any messages?

Your program runs a coroutine that never yields control to the event loop. The coroutine that receives messages never gets a chance to run.

Putting an `await` statement in a `for` or a `while` loop isn't enough to yield control. Awaiting a coroutine may yield control, but there's no guarantee that it will.

For example, *`send()`* only yields control when send buffers are full, which never happens in most practical cases.

If you run a loop that contains only synchronous operations and a *`send()`* call, you must yield control explicitly with `asyncio.sleep()`:

```
async def producer(websocket):
    message = generate_next_message()
    await websocket.send(message)
    await asyncio.sleep(0)  # yield control to the event loop
```

`asyncio.sleep()` always suspends the current task, allowing other tasks to run. This behavior is documented precisely because it isn't expected from every coroutine.

See issue 867.

Why does my very simple program misbehave mysteriously?

You are using `time.sleep()` instead of `asyncio.sleep()`, which blocks the event loop and prevents asyncio from operating normally.

This may lead to messages getting send but not received, to connection timeouts, and to unexpected results of shotgun debugging e.g. adding an unnecessary call to *`send()`* makes the program functional.

2.1.4 Both sides

What does `ConnectionClosedError: no close frame received or sent` mean?

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

```
connection handler failed
Traceback (most recent call last):
  ...
asyncio.exceptions.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):
```

(continues on next page)

(continued from previous page)

```
  ...
websockets.exceptions.ConnectionClosedError: no close frame received or sent
```

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: no close frame received or sent
```

it means that the TCP connection was lost. As a consequence, the WebSocket connection was closed without receiving and sending 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, despite websockets' keepalive mechanism.

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

What does `ConnectionClosedError: sent 1011 (unexpected error) keepalive ping timeout; no close frame received` mean?

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

```
connection handler failed
Traceback (most recent call last):
  ...
asyncio.exceptions.CancelledError

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

Traceback (most recent call last):
  ...
websockets.exceptions.ConnectionClosedError: sent 1011 (unexpected error) keepalive ping␣
↪timeout; no close frame received
```

or if a client crashes with this traceback:

```
Traceback (most recent call last):
  ...
asyncio.exceptions.CancelledError

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

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  ...
websockets.exceptions.ConnectionClosedError: sent 1011 (unexpected error) keepalive ping␣
↪timeout; no close frame received
```

it means that the WebSocket connection suffered from excessive latency and was closed after reaching the timeout of websockets' keepalive mechanism.

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

There are two main reasons why latency may increase:

- Poor network connectivity.
- More traffic than the recipient can handle.

See the discussion of *timeouts* for details.

If websockets' default timeout of 20 seconds is too short for your use case, you can adjust it with the `ping_timeout` argument.

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 can I pass additional arguments to a custom protocol subclass?

You can bind additional arguments to the protocol factory with `functools.partial()`:

```
import asyncio
import functools
import websockets

class MyServerProtocol(websockets.WebSocketServerProtocol):
    def __init__(self, extra_argument, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # do something with extra_argument

create_protocol = functools.partial(MyServerProtocol, extra_argument='spam')
start_server = websockets.serve(..., create_protocol=create_protocol)
```

This example was for a server. The same pattern applies on a client.

How do I keep idle connections open?

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

It 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.

2.1.5 Miscellaneous

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.

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(...)`. Unfortunately, this is deprecated as of Python 3.10.

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

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.

Is there a Python 2 version?

No, there isn't.

Python 2 reached end of life on January 1st, 2020.

Before that date, websockets required asyncio and therefore Python 3.

Why do I get the error: `module 'websockets' has no attribute '...'`?

Often, this is because you created a script called `websockets.py` in your current working directory. Then `import websockets` imports this module instead of the websockets library.

I'm having problems with threads

You shouldn't use threads. Use tasks instead.

`call_soon_threadsafe()` may help.

2.2 Cheat sheet

2.2.1 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.

2.2.2 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.

2.2.3 Debugging

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

```
import logging
logger = logging.getLogger('websockets')
logger.setLevel(logging.DEBUG)
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!

This guide will help you integrate websockets into a broader system.

2.3 Integrate with Django

If you're looking at adding real-time capabilities to a Django project with WebSocket, you have two main options.

1. Using Django Channels, a project adding WebSocket to Django, among other features. This approach is fully supported by Django. However, it requires switching to a new deployment architecture.
2. Deploying a separate WebSocket server next to your Django project. This technique is well suited when you need to add a small set of real-time features — maybe a notification service — to a HTTP application.

This guide shows how to implement the second technique with websockets. It assumes familiarity with Django.

2.3.1 Authenticate connections

Since the websockets server runs outside of Django, we need to integrate it with `django.contrib.auth`.

We will generate authentication tokens in the Django project. Then we will send them to the websockets server, where they will authenticate the user.

Generating a token for the current user and making it available in the browser is up to you. You could render the token in a template or fetch it with an API call.

Refer to the topic guide on *authentication* for details on this design.

Generate tokens

We want secure, short-lived tokens containing the user ID. We'll rely on django-sesame, a small library designed exactly for this purpose.

Add django-sesame to the dependencies of your Django project, install it, and configure it in the settings of the project:

```
AUTHENTICATION_BACKENDS = [
    "django.contrib.auth.backends.ModelBackend",
    "sesame.backends.ModelBackend",
]
```

(If your project already uses another authentication backend than the default `"django.contrib.auth.backends.ModelBackend"`, adjust accordingly.)

You don't need `"sesame.middleware.AuthenticationMiddleware"`. It is for authenticating users in the Django server, while we're authenticating them in the websockets server.

We'd like our tokens to be valid for 30 seconds. We expect web pages to load and to establish the WebSocket connection within this delay. Configure django-sesame accordingly in the settings of your Django project:

```
SESAME_MAX_AGE = 30
```

If you expect your web site to load faster for all clients, a shorter lifespan is possible. However, in the context of this document, it would make manual testing more difficult.

You could also enable single-use tokens. However, this would update the last login date of the user every time a WebSocket connection is established. This doesn't seem like a good idea, both in terms of behavior and in terms of performance.

Now you can generate tokens in a `django-admin shell` as follows:

```
>>> from django.contrib.auth import get_user_model
>>> User = get_user_model()
>>> user = User.objects.get(username="<your username>")
>>> from sesame.utils import get_token
>>> get_token(user)
'<your token>'
```

Keep this console open: since tokens expire after 30 seconds, you'll have to generate a new token every time you want to test connecting to the server.

Validate tokens

Let's move on to the websockets server.

Add websockets to the dependencies of your Django project and install it. Indeed, we're going to reuse the environment of the Django project, so we can call its APIs in the websockets server.

Now here's how to implement authentication.

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

import asyncio

```

(continues on next page)

(continued from previous page)

```
import django
import websockets

django.setup()

from sesame.utils import get_user


async def handler(websocket, path):
    sesame = await websocket.recv()
    user = await asyncio.to_thread(get_user, sesame)
    if user is None:
        await websocket.close(1011, "authentication failed")
        return

    await websocket.send(f"Hello {user}!")


async def main():
    async with websockets.serve(handler, "localhost", 8888):
        await asyncio.Future()  # run forever


if __name__ == "__main__":
    asyncio.run(main())
```

Let's unpack this code.

We're calling `django.setup()` before doing anything with Django because we're using Django in a standalone script. This assumes that the `DJANGO_SETTINGS_MODULE` environment variable is set to the Python path to your settings module.

The connection handler reads the first message received from the client, which is expected to contain a django-sesame token. Then it authenticates the user with `get_user()`, the API for authentication outside views. If authentication fails, it closes the connection and exits.

When we call an API that makes a database query such as `get_user()`, we wrap the call in `to_thread()`. Indeed, the Django ORM doesn't support asynchronous I/O. It would block the event loop if it didn't run in a separate thread. `to_thread()` is available since Python 3.9. In earlier versions, use `run_in_executor()` instead.

Finally, we start a server with *`serve()`*.

We're ready to test!

Save this code to a file called `authentication.py`, make sure the `DJANGO_SETTINGS_MODULE` environment variable is set properly, and start the websockets server:

```
$ python authentication.py
```

Generate a new token — remember, they're only valid for 30 seconds — and use it to connect to your server. Paste your token and press Enter when you get a prompt:

```
$ python -m websockets ws://localhost:8888/
Connected to ws://localhost:8888/
> <your token>
```

(continues on next page)

(continued from previous page)

```
< Hello <your username>!
Connection closed: 1000 (OK).
```

It works!

If you enter an expired or invalid token, authentication fails and the server closes the connection:

```
$ python -m websockets ws://localhost:8888/
Connected to ws://localhost:8888.
> not a token
Connection closed: 1011 (unexpected error) authentication failed.
```

You can also test from a browser by generating a new token and running the following code in the JavaScript console of the browser:

```
websocket = new WebSocket("ws://localhost:8888/");
websocket.onopen = (event) => websocket.send("<your token>");
websocket.onmessage = (event) => console.log(event.data);
```

If you don't want to import your entire Django project into the websockets server, you can build a separate Django project with `django.contrib.auth`, `django-sesame`, a suitable `User` model, and a subset of the settings of the main project.

2.3.2 Stream events

We can connect and authenticate but our server doesn't do anything useful yet!

Let's send a message every time a user makes an action in the admin. This message will be broadcast to all users who can access the model on which the action was made. This may be used for showing notifications to other users.

Many use cases for WebSocket with Django follow a similar pattern.

Set up event bus

We need a event bus to enable communications between Django and websockets. Both sides connect permanently to the bus. Then Django writes events and websockets reads them. For the sake of simplicity, we'll rely on Redis Pub/Sub.

The easiest way to add Redis to a Django project is by configuring a cache backend with django-redis. This library manages connections to Redis efficiently, persisting them between requests, and provides an API to access the Redis connection directly.

Install Redis, add django-redis to the dependencies of your Django project, install it, and configure it in the settings of the project:

```
CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
    },
}
```

If you already have a default cache, add a new one with a different name and change `get_redis_connection("default")` in the code below to the same name.

Publish events

Now let's write events to the bus.

Add the following code to a module that is imported when your Django project starts. Typically, you would put it in a `signals.py` module, which you would import in the `AppConfig.ready()` method of one of your apps:

```
import json

from django.contrib.admin.models import LogEntry
from django.db.models.signals import post_save
from django.dispatch import receiver

from django_redis import get_redis_connection


@receiver(post_save, sender=LogEntry)
def publish_event(instance, **kwargs):
    event = {
        "model": instance.content_type.name,
        "object": instance.object_repr,
        "message": instance.get_change_message(),
        "timestamp": instance.action_time.isoformat(),
        "user": str(instance.user),
        "content_type_id": instance.content_type_id,
        "object_id": instance.object_id,
    }
    connection = get_redis_connection("default")
    payload = json.dumps(event)
    connection.publish("events", payload)
```

This code runs every time the admin saves a `LogEntry` object to keep track of a change. It extracts interesting data, serializes it to JSON, and writes an event to Redis.

Let's check that it works:

```
$ redis-cli
127.0.0.1:6379> SELECT 1
OK
127.0.0.1:6379[1]> SUBSCRIBE events
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "events"
3) (integer) 1
```

Leave this command running, start the Django development server and make changes in the admin: add, modify, or delete objects. You should see corresponding events published to the `"events"` stream.

Broadcast events

Now let's turn to reading events and broadcasting them to connected clients. We need to add several features:

- Keep track of connected clients so we can broadcast messages.
- Tell which content types the user has permission to view or to change.
- Connect to the message bus and read events.
- Broadcast these events to users who have corresponding permissions.

Here's a complete implementation.

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

import asyncio
import json

import aioredis
import django
import websockets

django.setup()

from django.contrib.contenttypes.models import ContentType
from sesame.utils import get_user


CONNECTIONS = {}


def get_content_types(user):
    """Return the set of IDs of content types visible by user."""
    # This does only three database queries because Django caches
    # all permissions on the first call to user.has_perm(...).
    return {
        ct.id
        for ct in ContentType.objects.all()
        if user.has_perm(f"{ct.app_label}.view_{ct.model}")
        or user.has_perm(f"{ct.app_label}.change_{ct.model}")
    }


async def handler(websocket, path):
    """Authenticate user and register connection in CONNECTIONS."""
    sesame = await websocket.recv()
    user = await asyncio.to_thread(get_user, sesame)
    if user is None:
        await websocket.close(1011, "authentication failed")
        return

    ct_ids = await asyncio.to_thread(get_content_types, user)
    CONNECTIONS[websocket] = {"content_type_ids": ct_ids}
    try:
        await websocket.wait_closed()
```

(continues on next page)

(continued from previous page)

```
    finally:
        del CONNECTIONS[websocket]


async def process_events():
    """Listen to events in Redis and process them."""
    redis = aioredis.from_url("redis://127.0.0.1:6379/1")
    pubsub = redis.pubsub()
    await pubsub.subscribe("events")
    async for message in pubsub.listen():
        if message["type"] != "message":
            continue
        payload = message["data"].decode()
        # Broadcast event to all users who have permissions to see it.
        event = json.loads(payload)
        recipients = (
            websocket
            for websocket, connection in CONNECTIONS.items()
            if event["content_type_id"] in connection["content_type_ids"]
        )
        websockets.broadcast(recipients, payload)


async def main():
    async with websockets.serve(handler, "localhost", 8888):
        await process_events()  # runs forever


if __name__ == "__main__":
    asyncio.run(main())
```

Since the `get_content_types()` function makes a database query, it is wrapped inside `asyncio.to_thread()`. It runs once when each WebSocket connection is open; then its result is cached for the lifetime of the connection. Indeed, running it for each message would trigger database queries for all connected users at the same time, which would hurt the database.

The connection handler merely registers the connection in a global variable, associated to the list of content types for which events should be sent to that connection, and waits until the client disconnects.

The `process_events()` function reads events from Redis and broadcasts them to all connections that should receive them. We don't care much if a sending a notification fails — this happens when a connection drops between the moment we iterate on connections and the moment the corresponding message is sent — so we start a task with for each message and forget about it. Also, this means we're immediately ready to process the next event, even if it takes time to send a message to a slow client.

Since Redis can publish a message to multiple subscribers, multiple instances of this server can safely run in parallel.

2.3.3 Does it scale?

In theory, given enough servers, this design can scale to a hundred million clients, since Redis can handle ten thousand servers and each server can handle ten thousand clients. In practice, you would need a more scalable message bus before reaching that scale, due to the volume of messages.

The WebSocket protocol makes provisions for extending or specializing its features, which websockets supports fully.

2.4 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 being sent or after being received.

As a consequence, writing an extension requires implementing several classes:

- Extension Factory: it negotiates parameters and instantiates the extension.

  Clients and servers require separate extension factories with distinct APIs.

  Extension factories are the public API of an extension.

- Extension: it decodes incoming frames and encodes outgoing frames.

  If the extension is symmetrical, clients and servers can use the same class.

  Extensions are initialized by extension factories, so they don't need to be part of the public API of an extension.

websockets provides base classes for extension factories and extensions. See *ClientExtensionFactory*, *ServerExtensionFactory*, and *Extension* for details. Once your application is ready, learn how to deploy it on various platforms.

2.5 Deploy to Heroku

This guide describes how to deploy a websockets server to Heroku. The same principles should apply to other Platform as a Service providers.

We're going to deploy a very simple app. The process would be identical for a more realistic app.

2.5.1 Create application

Deploying to Heroku requires a git repository. Let's initialize one:

```
$ mkdir websockets-echo
$ cd websockets-echo
$ git init -b main
Initialized empty Git repository in websockets-echo/.git/
$ git commit --allow-empty -m "Initial commit."
[main (root-commit) 1e7947d] Initial commit.
```

Follow the set-up instructions to install the Heroku CLI and to log in, if you haven't done that yet.

Then, create a Heroku app — if you follow these instructions step-by-step, you'll have to pick a different name because I'm already using `websockets-echo` on Heroku:

```
$ heroku create websockets-echo
Creating  websockets-echo... done
https://websockets-echo.herokuapp.com/ | https://git.heroku.com/websockets-echo.git
```

Here's the implementation of the app, an echo server. Save it in a file called `app.py`:

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

import asyncio
import signal
import os

import websockets


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


async def main():
    # Set the stop condition when receiving SIGTERM.
    loop = asyncio.get_running_loop()
    stop = loop.create_future()
    loop.add_signal_handler(signal.SIGTERM, stop.set_result, None)

    async with websockets.serve(
        echo,
        host="",
        port=int(os.environ["PORT"]),
    ):
        await stop


if __name__ == "__main__":
    asyncio.run(main())
```

Heroku expects the server to listen on a specific port, which is provided in the `$PORT` environment variable. The app reads it and passes it to *`serve()`*.

Heroku sends a `SIGTERM` signal to all processes when shutting down a dyno. When the app receives this signal, it closes connections and exits cleanly.

2.5.2 Deploy application

In order to build the app, Heroku needs to know that it depends on websockets. Create a `requirements.txt` file containing this line:

```
websockets
```

Heroku also needs to know how to run the app. Create a `Procfile` with this content:

```
web: python app.py
```

Confirm that you created the correct files and commit them to git:

```
$ ls
Procfile         app.py           requirements.txt
$ git add .
$ git commit -m "Deploy echo server to Heroku."
[main 8418c62] Deploy echo server to Heroku.
 3 files changed, 32 insertions(+)
 create mode 100644 Procfile
 create mode 100644 app.py
 create mode 100644 requirements.txt
```

The app is ready. Let's deploy it!

```
$ git push heroku main

... lots of output...

remote: -----> Launching...
remote:        Released v1
remote:        https://websockets-echo.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/websockets-echo.git
 * [new branch]      main -> main
```

2.5.3 Validate deployment

Of course you'd like to confirm that your application is running as expected!

Since it's a WebSocket server, you need a WebSocket client, such as the interactive client that comes with websockets.

If you're currently building a websockets server, perhaps you're already in a virtualenv where websockets is installed. If not, you can install it in a new virtualenv as follows:

```
$ python -m venv websockets-client
$ . websockets-client/bin/activate
$ pip install websockets
```

Connect the interactive client — using the name of your Heroku app instead of `websockets-echo`:

```
$ python -m websockets wss://websockets-echo.herokuapp.com/
Connected to wss://websockets-echo.herokuapp.com/.
>
```

Great! Your app is running!

In this example, I used a secure connection (`wss://`). It worked because Heroku served a valid TLS certificate for `websockets-echo.herokuapp.com`. An insecure connection (`ws://`) would also work.

Once you're connected, you can send any message and the server will echo it, then press Ctrl-D to terminate the connection:

```
> Hello!
< Hello!
Connection closed: 1000 (OK).
```

You can also confirm that your application shuts down gracefully. Connect an interactive client again — remember to replace `websockets-echo` with your app:

```
$ python -m websockets wss://websockets-echo.herokuapp.com/
Connected to wss://websockets-echo.herokuapp.com/.
>
```

In another shell, restart the dyno — again, replace `websockets-echo` with your app:

```
$ heroku dyno:restart -a websockets-echo
Restarting dynos on  websockets-echo... done
```

Go back to the first shell. The connection is closed with code 1001 (going away).

```
$ python -m websockets wss://websockets-echo.herokuapp.com/
Connected to wss://websockets-echo.herokuapp.com/.
Connection closed: 1001 (going away).
```

If graceful shutdown wasn't working, the server wouldn't perform a closing handshake and the connection would be closed with code 1006 (connection closed abnormally).

2.6 Deploy to Kubernetes

This guide describes how to deploy a websockets server to Kubernetes. It assumes familiarity with Docker and Kubernetes.

We're going to deploy a simple app to a local Kubernetes cluster and to ensure that it scales as expected.

In a more realistic context, you would follow your organization's practices for deploying to Kubernetes, but you would apply the same principles as far as websockets is concerned.

2.6.1 Containerize application

Here's the app we're going to deploy. Save it in a file called `app.py`:

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

import asyncio
import http
import signal
import sys
import time

import websockets


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

(continues on next page)

(continued from previous page)

```
        # Block the event loop! This allows saturating a single asyncio
        # process without opening an impractical number of connections.
        time.sleep(0.1)  # 100ms
        await websocket.send(message)


async def health_check(path, request_headers):
    if path == "/healthz":
        return http.HTTPStatus.OK, [], b"OK\n"
    if path == "/inemuri":
        loop = asyncio.get_running_loop()
        loop.call_later(1, time.sleep, 10)
        return http.HTTPStatus.OK, [], b"Sleeping for 10s\n"
    if path == "/seppuku":
        loop = asyncio.get_running_loop()
        loop.call_later(1, sys.exit, 69)
        return http.HTTPStatus.OK, [], b"Terminating\n"


async def main():
    # Set the stop condition when receiving SIGTERM.
    loop = asyncio.get_running_loop()
    stop = loop.create_future()
    loop.add_signal_handler(signal.SIGTERM, stop.set_result, None)

    async with websockets.serve(
        slow_echo,
        host="",
        port=80,
        process_request=health_check,
    ):
        await stop


if __name__ == "__main__":
    asyncio.run(main())
```

This is an echo server with one twist: every message blocks the server for 100ms, which creates artificial starvation of CPU time. This makes it easier to saturate the server for load testing.

The app exposes a health check on `/healthz`. It also provides two other endpoints for testing purposes: `/inemuri` will make the app unresponsive for 10 seconds and `/seppuku` will terminate it.

The quest for the perfect Python container image is out of scope of this guide, so we'll go for the simplest possible configuration instead:

```
FROM python:3.9-alpine

RUN pip install websockets

COPY app.py .

CMD ["python", "app.py"]
```

After saving this `Dockerfile`, build the image:

```
$ docker build -t websockets-test:1.0 .
```

Test your image by running:

```
$ docker run --name run-websockets-test --publish 32080:80 --rm \
    websockets-test:1.0
```

Then, in another shell, in a virtualenv where websockets is installed, connect to the app and check that it echoes anything you send:

```
$ python -m websockets ws://localhost:32080/
Connected to ws://localhost:32080/.
> Hey there!
< Hey there!
>
```

Now, in yet another shell, stop the app with:

```
$ docker kill -s TERM run-websockets-test
```

Going to the shell where you connected to the app, you can confirm that it shut down gracefully:

```
$ python -m websockets ws://localhost:32080/
Connected to ws://localhost:32080/.
> Hey there!
< Hey there!
Connection closed: 1001 (going away).
```

If it didn't, you'd get code 1006 (connection closed abnormally).

2.6.2 Deploy application

Configuring Kubernetes is even further beyond the scope of this guide, so we'll use a basic configuration for testing, with just one Service and one Deployment:

```
apiVersion: v1
kind: Service
metadata:
  name: websockets-test
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 32080
  selector:
    app: websockets-test
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: websockets-test
```

(continues on next page)

(continued from previous page)

```
spec:
  selector:
    matchLabels:
      app: websockets-test
  template:
    metadata:
      labels:
        app: websockets-test
    spec:
      containers:
      - name: websockets-test
        image: websockets-test:1.0
        livenessProbe:
          httpGet:
            path: /healthz
            port: 80
          periodSeconds: 1
        ports:
        - containerPort: 80
```

For local testing, a service of type NodePort is good enough. For deploying to production, you would configure an Ingress.

After saving this to a file called `deployment.yaml`, you can deploy:

```
$ kubectl apply -f deployment.yaml
service/websockets-test created
deployment.apps/websockets-test created
```

Now you have a deployment with one pod running:

```
$ kubectl get deployment websockets-test
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
websockets-test   1/1     1            1           10s
$ kubectl get pods -l app=websockets-test
NAME                               READY   STATUS    RESTARTS   AGE
websockets-test-86b48f4bb7-nltfh   1/1     Running   0          10s
```

You can connect to the service — press Ctrl-D to exit:

```
$ python -m websockets ws://localhost:32080/
Connected to ws://localhost:32080/.
Connection closed: 1000 (OK).
```

2.6.3 Validate deployment

First, let's ensure the liveness probe works by making the app unresponsive:

```
$ curl http://localhost:32080/inemuri
Sleeping for 10s
```

Since we have only one pod, we know that this pod will go to sleep.

The liveness probe is configured to run every second. By default, liveness probes time out after one second and have a threshold of three failures. Therefore Kubernetes should restart the pod after at most 5 seconds.

Indeed, after a few seconds, the pod reports a restart:

```
$ kubectl get pods -l app=websockets-test
NAME                                READY   STATUS    RESTARTS   AGE
websockets-test-86b48f4bb7-nltfh    1/1     Running   1          42s
```

Next, let's take it one step further and crash the app:

```
$ curl http://localhost:32080/seppuku
Terminating
```

The pod reports a second restart:

```
$ kubectl get pods -l app=websockets-test
NAME                                READY   STATUS    RESTARTS   AGE
websockets-test-86b48f4bb7-nltfh    1/1     Running   2          72s
```

All good — Kubernetes delivers on its promise to keep our app alive!

2.6.4 Scale deployment

Of course, Kubernetes is for scaling. Let's scale — modestly — to 10 pods:

```
$ kubectl scale deployment.apps/websockets-test --replicas=10
deployment.apps/websockets-test scaled
```

After a few seconds, we have 10 pods running:

```
$ kubectl get deployment websockets-test
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
websockets-test   10/10   10           10          10m
```

Now let's generate load. We'll use this script:

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

import asyncio
import sys
import websockets


URI = "ws://localhost:32080"
```

(continues on next page)

(continued from previous page)

```
async def run(client_id, messages):
    async with websockets.connect(URI) as websocket:
        for message_id in range(messages):
            await websocket.send("{client_id}:{message_id}")
            await websocket.recv()


async def benchmark(clients, messages):
    await asyncio.wait([
        asyncio.create_task(run(client_id, messages))
        for client_id in range(clients)
    ])


if __name__ == "__main__":
    clients, messages = int(sys.argv[1]), int(sys.argv[2])
    asyncio.run(benchmark(clients, messages))
```

We'll connect 500 clients in parallel, meaning 50 clients per pod, and have each client send 6 messages. Since the app blocks for 100ms before responding, if connections are perfectly distributed, we expect a total run time slightly over 50 * 6 * 0.1 = 30 seconds.

Let's try it:

```
$ ulimit -n 512
$ time python benchmark.py 500 6
python benchmark.py 500 6  2.40s user 0.51s system 7% cpu 36.471 total
```

A total runtime of 36 seconds is in the right ballpark. Repeating this experiment with other parameters shows roughly consistent results, with the high variability you'd expect from a quick benchmark without any effort to stabilize the test setup.

Finally, we can scale back to one pod.

```
$ kubectl scale deployment.apps/websockets-test --replicas=1
deployment.apps/websockets-test scaled
$ kubectl get deployment websockets-test
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
websockets-test   1/1     1            1           15m
```

2.7 Deploy with Supervisor

This guide proposes a simple way to deploy a websockets server directly on a Linux or BSD operating system.

We'll configure Supervisor to run several server processes and to restart them if needed.

We'll bind all servers to the same port. The OS will take care of balancing connections.

Create and activate a virtualenv:

```
$ python -m venv supervisor-websockets
$ . supervisor-websockets/bin/activate
```

Install websockets and Supervisor:

```
$ pip install websockets
$ pip install supervisor
```

Save this app to a file called `app.py`:

```
#!/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 main():
    # Set the stop condition when receiving SIGTERM.
    loop = asyncio.get_running_loop()
    stop = loop.create_future()
    loop.add_signal_handler(signal.SIGTERM, stop.set_result, None)

    async with websockets.serve(
        echo,
        host="",
        port=8080,
        reuse_port=True,
    ):
        await stop


if __name__ == "__main__":
    asyncio.run(main())
```

This is an echo server with two features added for the purpose of this guide:

- It shuts down gracefully when receiving a `SIGTERM` signal;
- It enables the `reuse_port` option of `create_server()`, which in turns sets `SO_REUSEPORT` on the accept socket.

Save this Supervisor configuration to `supervisord.conf`:

```
[supervisord]

[program:websockets-test]
command = python app.py
process_name = %(program_name)s_%(process_num)02d
numprocs = 4
autorestart = true
```

This is the minimal configuration required to keep four instances of the app running, restarting them if they exit.

Now start Supervisor in the foreground:

```
$ supervisord -c supervisord.conf -n
INFO Increased RLIMIT_NOFILE limit to 1024
INFO supervisord started with pid 43596
INFO spawned: 'websockets-test_00' with pid 43597
INFO spawned: 'websockets-test_01' with pid 43598
INFO spawned: 'websockets-test_02' with pid 43599
INFO spawned: 'websockets-test_03' with pid 43600
INFO success: websockets-test_00 entered RUNNING state, process has stayed up for > than␣
↪1 seconds (startsecs)
INFO success: websockets-test_01 entered RUNNING state, process has stayed up for > than␣
↪1 seconds (startsecs)
INFO success: websockets-test_02 entered RUNNING state, process has stayed up for > than␣
↪1 seconds (startsecs)
INFO success: websockets-test_03 entered RUNNING state, process has stayed up for > than␣
↪1 seconds (startsecs)
```

In another shell, after activating the virtualenv, we can connect to the app — press Ctrl-D to exit:

```
$ python -m websockets ws://localhost:8080/
Connected to ws://localhost:8080/.
> Hello!
< Hello!
Connection closed: 1000 (OK).
```

Look at the pid of an instance of the app in the logs and terminate it:

```
$ kill -TERM 43597
```

The logs show that Supervisor restarted this instance:

```
INFO exited: websockets-test_00 (exit status 0; expected)
INFO spawned: 'websockets-test_00' with pid 43629
INFO success: websockets-test_00 entered RUNNING state, process has stayed up for > than␣
↪1 seconds (startsecs)
```

Now let's check what happens when we shut down Supervisor, but first let's establish a connection and leave it open:

```
$ python -m websockets ws://localhost:8080/
Connected to ws://localhost:8080/.
>
```

Look at the pid of supervisord itself in the logs and terminate it:

```
$ kill -TERM 43596
```

The logs show that Supervisor terminated all instances of the app before exiting:

```
WARN received SIGTERM indicating exit request
INFO waiting for websockets-test_00, websockets-test_01, websockets-test_02, websockets-
↪test_03 to die
INFO stopped: websockets-test_02 (exit status 0)
INFO stopped: websockets-test_03 (exit status 0)
```

(continues on next page)

(continued from previous page)

```
INFO stopped: websockets-test_01 (exit status 0)
INFO stopped: websockets-test_00 (exit status 0)
```

And you can see that the connection to the app was closed gracefully:

```
$ python -m websockets ws://localhost:8080/
Connected to ws://localhost:8080/.
Connection closed: 1001 (going away).
```

In this example, we've been sharing the same virtualenv for supervisor and websockets.

In a real deployment, you would likely:

- Install Supervisor with the package manager of the OS.
- Create a virtualenv dedicated to your application.
- Add `environment=PATH="path/to/your/virtualenv/bin"` in the Supervisor configuration. Then `python app.py` runs in that virtualenv.

2.8 Deploy behind nginx

This guide demonstrates a way to load balance connections across multiple websockets server processes running on the same machine with nginx.

We'll run server processes with Supervisor as described in *this guide*.

2.8.1 Run server processes

Save this app to `app.py`:

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

import asyncio
import os
import signal

import websockets


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


async def main():
    # Set the stop condition when receiving SIGTERM.
    loop = asyncio.get_running_loop()
    stop = loop.create_future()
    loop.add_signal_handler(signal.SIGTERM, stop.set_result, None)

    async with websockets.unix_serve(
```

(continues on next page)

(continued from previous page)

```
        echo,
        path=f"{os.environ['SUPERVISOR_PROCESS_NAME']}.sock",
    ):
        await stop


if __name__ == "__main__":
    asyncio.run(main())
```

We'd like to nginx to connect to websockets servers via Unix sockets in order to avoid the overhead of TCP for communicating between processes running in the same OS.

We start the app with *unix_serve()*. Each server process listens on a different socket thanks to an environment variable set by Supervisor to a different value.

Save this configuration to `supervisord.conf`:

```
[supervisord]

[program:websockets-test]
command = python app.py
process_name = %(program_name)s_%(process_num)02d
numprocs = 4
autorestart = true
```

This configuration runs four instances of the app.

Install Supervisor and run it:

```
$ supervisord -c supervisord.conf -n
```

2.8.2 Configure and run nginx

Here's a simple nginx configuration to load balance connections across four processes:

```
daemon off;

events {
}

http {
    server {
        listen localhost:8080;

        location / {
            proxy_http_version 1.1;
            proxy_pass http://websocket;
            proxy_set_header Connection $http_connection;
            proxy_set_header Upgrade $http_upgrade;
        }
    }

    upstream websocket {
```

(continues on next page)

(continued from previous page)

```
        least_conn;
        server unix:websockets-test_00.sock;
        server unix:websockets-test_01.sock;
        server unix:websockets-test_02.sock;
        server unix:websockets-test_03.sock;
    }
}
```

We set `daemon off` so we can run nginx in the foreground for testing.

Then we combine the WebSocket proxying and load balancing guides:

- The WebSocket protocol requires HTTP/1.1. We must set the HTTP protocol version to 1.1, else nginx defaults to HTTP/1.0 for proxying.
- The WebSocket handshake involves the `Connection` and `Upgrade` HTTP headers. We must pass them to the upstream explicitly, else nginx drops them because they're hop-by-hop headers.

  We deviate from the WebSocket proxying guide because its example adds a `Connection: Upgrade` header to every upstream request, even if the original request didn't contain that header.
- In the upstream configuration, we set the load balancing method to `least_conn` in order to balance the number of active connections across servers. This is best for long running connections.

Save the configuration to `nginx.conf`, install nginx, and run it:

```
$ nginx -c nginx.conf -p .
```

You can confirm that nginx proxies connections properly:

```
$ PYTHONPATH=src python -m websockets ws://localhost:8080/
Connected to ws://localhost:8080/.
> Hello!
< Hello!
Connection closed: 1000 (OK).
```

2.9 Deploy behind HAProxy

This guide demonstrates a way to load balance connections across multiple websockets server processes running on the same machine with HAProxy.

We'll run server processes with Supervisor as described in *this guide*.

2.9.1 Run server processes

Save this app to `app.py`:

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

import asyncio
import os
import signal
```

(continues on next page)

(continued from previous page)

```
import websockets


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


async def main():
    # Set the stop condition when receiving SIGTERM.
    loop = asyncio.get_running_loop()
    stop = loop.create_future()
    loop.add_signal_handler(signal.SIGTERM, stop.set_result, None)

    async with websockets.serve(
        echo,
        host="localhost",
        port=8000 + int(os.environ["SUPERVISOR_PROCESS_NAME"][-2:]),
    ):
        await stop


if __name__ == "__main__":
    asyncio.run(main())
```

Each server process listens on a different port by extracting an incremental index from an environment variable set by Supervisor.

Save this configuration to `supervisord.conf`:

```
[supervisord]

[program:websockets-test]
command = python app.py
process_name = %(program_name)s_%(process_num)02d
numprocs = 4
autorestart = true
```

This configuration runs four instances of the app.

Install Supervisor and run it:

```
$ supervisord -c supervisord.conf -n
```

2.9.2 Configure and run HAProxy

Here's a simple HAProxy configuration to load balance connections across four processes:

```
defaults
    mode http
    timeout connect 10s
    timeout client 30s
    timeout server 30s

frontend websocket
    bind localhost:8080
    default_backend websocket

backend websocket
    balance leastconn
    server websockets-test_00 localhost:8000
    server websockets-test_01 localhost:8001
    server websockets-test_02 localhost:8002
    server websockets-test_03 localhost:8003
```

In the backend configuration, we set the load balancing method to `leastconn` in order to balance the number of active connections across servers. This is best for long running connections.

Save the configuration to `haproxy.cfg`, install HAProxy, and run it:

```
$ haproxy -f haproxy.cfg
```

You can confirm that HAProxy proxies connections properly:

```
$ PYTHONPATH=src python -m websockets ws://localhost:8080/
Connected to ws://localhost:8080/.
> Hello!
< Hello!
Connection closed: 1000 (OK).
```

If you're integrating the Sans-I/O layer of websockets into a library, rather than building an application with websockets, follow this guide.

2.10 Integrate the Sans-I/O layer

This guide explains how to integrate the Sans-I/O layer of websockets to add support for WebSocket in another library.

As a prerequisite, you should decide how you will handle network I/O and asynchronous control flow.

Your integration layer will provide an API for the application on one side, will talk to the network on the other side, and will rely on websockets to implement the protocol in the middle.

2.10.1 Opening a connection

Client-side

If you're building a client, parse the URI you'd like to connect to:

```
from websockets.uri import parse_uri

wsuri = parse_uri("ws://example.com/")
```

Open a TCP connection to `(wsuri.host, wsuri.port)` and perform a TLS handshake if `wsuri.secure` is `True`.

Initialize a *ClientConnection*:

```
from websockets.client import ClientConnection

connection = ClientConnection(wsuri)
```

Create a WebSocket handshake request with *connect()* and send it with *send_request()*:

```
request = connection.connect()
connection.send_request(request)
```

Then, call *data_to_send()* and send its output to the network, as described in *Send data* below.

The first event returned by *events_received()* is the WebSocket handshake response.

When the handshake fails, the reason is available in `response.exception`:

```
if response.exception is not None:
    raise response.exception
```

Else, the WebSocket connection is open.

A WebSocket client API usually performs the handshake then returns a wrapper around the network connection and the *ClientConnection*.

Server-side

If you're building a server, accept network connections from clients and perform a TLS handshake if desired.

For each connection, initialize a *ServerConnection*:

```
from websockets.server import ServerConnection

connection = ServerConnection()
```

The first event returned by *events_received()* is the WebSocket handshake request.

Create a WebSocket handshake response with *accept()* and send it with *send_response()*:

```
response = connection.accept(request)
connection.send_response(response)
```

Alternatively, you may reject the WebSocket handshake and return a HTTP response with *reject()*:

```
response = connection.reject(status, explanation)
connection.send_response(response)
```

Then, call *data_to_send()* and send its output to the network, as described in *Send data* below.

Even when you call *accept()*, the WebSocket handshake may fail if the request is incorrect or unsupported.

When the handshake fails, the reason is available in `request.exception`:

```
if request.exception is not None:
    raise request.exception
```

Else, the WebSocket connection is open.

A WebSocket server API usually builds a wrapper around the network connection and the *ServerConnection*. Then it invokes a connection handler that accepts the wrapper in argument.

It may also provide a way to close all connections and to shut down the server gracefully.

Going forwards, this guide focuses on handling an individual connection.

2.10.2 From the network to the application

Go through the five steps below until you reach the end of the data stream.

Receive data

When receiving data from the network, feed it to the connection's *receive_data()* method.

When reaching the end of the data stream, call the connection's *receive_eof()* method.

For example, if `sock` is a `socket`:

```
try:
    data = sock.recv(4096)
except OSError:  # socket closed
    data = b""
if data:
    connection.receive_data(data)
else:
    connection.receive_eof()
```

These methods aren't expected to raise exceptions — unless you call them again after calling *receive_eof()*, which is an error. (If you get an exception, please file a bug!)

Send data

Then, call *data_to_send()* and send its output to the network:

```
for data in connection.data_to_send():
    if data:
        sock.sendall(data)
    else:
        sock.shutdown(socket.SHUT_WR)
```

The empty bytestring signals the end of the data stream. When you see it, you must half-close the TCP connection.

Sending data right after receiving data is necessary because websockets responds to ping frames, close frames, and incorrect inputs automatically.

Expect TCP connection to close

Closing a WebSocket connection normally involves a two-way WebSocket closing handshake. Then, regardless of whether the closure is normal or abnormal, the server starts the four-way TCP closing handshake. If the network fails at the wrong point, you can end up waiting until the TCP timeout, which is very long.

To prevent dangling TCP connections when you expect the end of the data stream but you never reach it, call *close_expected()* and, if it returns `True`, schedule closing the TCP connection after a short timeout:

```
# start a new execution thread to run this code
sleep(10)
sock.close()  # does nothing if the socket is already closed
```

If the connection is still open when the timeout elapses, closing the socket makes the execution thread that reads from the socket reach the end of the data stream, possibly with an exception.

Close TCP connection

If you called *receive_eof()*, close the TCP connection now. This is a clean closure because the receive buffer is empty.

After *receive_eof()* signals the end of the read stream, *data_to_send()* always signals the end of the write stream, unless it already ended. So, at this point, the TCP connection is already half-closed. The only reason for closing it now is to release resources related to the socket.

Now you can exit the loop relaying data from the network to the application.

Receive events

Finally, call *events_received()* to obtain events parsed from the data provided to *receive_data()*:

```
events = connection.events_received()
```

The first event will be the WebSocket opening handshake request or response. See *Opening a connection* above for details.

All later events are WebSocket frames. There are two types of frames:

- Data frames contain messages transferred over the WebSocket connections. You should provide them to the application. See *Fragmentation* below for how to reassemble messages from frames.
- Control frames provide information about the connection's state. The main use case is to expose an abstraction over ping and pong to the application. Keep in mind that websockets responds to ping frames and close frames automatically. Don't duplicate this functionality!

2.10.3 From the application to the network

The connection object provides one method for each type of WebSocket frame.

For sending a data frame:

- *send_continuation()*
- *send_text()*
- *send_binary()*

These methods raise *ProtocolError* if you don't set the *FIN* bit correctly in fragmented messages.

For sending a control frame:

- *send_close()*
- *send_ping()*
- *send_pong()*

send_close() initiates the closing handshake. See *Closing a connection* below for details.

If you encounter an unrecoverable error and you must fail the WebSocket connection, call *fail()*.

After any of the above, call *data_to_send()* and send its output to the network, as shown in *Send data* above.

If you called *send_close()* or *fail()*, you expect the end of the data stream. You should follow the process described in *Close TCP connection* above in order to prevent dangling TCP connections.

2.10.4 Closing a connection

Under normal circumstances, when a server wants to close the TCP connection:

- it closes the write side;
- it reads until the end of the stream, because it expects the client to close the read side;
- it closes the socket.

When a client wants to close the TCP connection:

- it reads until the end of the stream, because it expects the server to close the read side;
- it closes the write side;
- it closes the socket.

Applying the rules described earlier in this document gives the intended result. As a reminder, the rules are:

- When *data_to_send()* returns the empty bytestring, close the write side of the TCP connection.
- When you reach the end of the read stream, close the TCP connection.
- When *close_expected()* returns `True`, if you don't reach the end of the read stream quickly, close the TCP connection.

2.10.5 Fragmentation

WebSocket messages may be fragmented. Since this is a protocol-level concern, you may choose to reassemble fragmented messages before handing them over to the application.

To reassemble a message, read data frames until you get a frame where the *FIN* bit is set, then concatenate the payloads of all frames.

You will never receive an inconsistent sequence of frames because websockets raises a *ProtocolError* and fails the connection when this happens. However, you may receive an incomplete sequence if the connection drops in the middle of a fragmented message.

2.10.6 Tips

Serialize operations

The Sans-I/O layer expects to run sequentially. If your interact with it from multiple threads or coroutines, you must ensure correct serialization. This should happen automatically in a cooperative multitasking environment.

However, you still have to make sure you don't break this property by accident. For example, serialize writes to the network when *data_to_send()* returns multiple values to prevent concurrent writes from interleaving incorrectly.

Avoid buffers

The Sans-I/O layer doesn't do any buffering. It makes events available in *events_received()* as soon as they're received.

You should make incoming messages available to the application immediately and stop further processing until the application fetches them. This will usually result in the best performance.

CHAPTER

THREE

API REFERENCE

websockets provides client and server implementations, as shown in the *getting started guide*.

The process for opening and closing a WebSocket connection depends on which side you're implementing.

- On the client side, connecting to a server with `connect()` yields a connection object that provides methods for interacting with the connection. Your code can open a connection, then send or receive messages.

  If you use `connect()` as an asynchronous context manager, then websockets closes the connection on exit. If not, then your code is responsible for closing the connection.

- On the server side, `serve()` starts listening for client connections and yields an server object that you can use to shut down the server.

  Then, when a client connects, the server initializes a connection object and passes it to a handler coroutine, which is where your code can send or receive messages. This pattern is called inversion of control. It's common in frameworks implementing servers.

  When the handler coroutine terminates, websockets closes the connection. You may also close it in the handler coroutine if you'd like.

Once the connection is open, the WebSocket protocol is symmetrical, except for low-level details that websockets manages under the hood. The same methods are available on client connections created with `connect()` and on server connections received in argument by the connection handler of `serve()`.

Since websockets provides the same API — and uses the same code — for client and server connections, common methods are documented in a "Both sides" page.

3.1 Client

3.1.1 asyncio

Opening a connection

await websockets.client.**connect**(*uri*, ***, *create_protocol=None*, *logger=None*, *compression='deflate'*, *origin=None*, *extensions=None*, *subprotocols=None*, *extra_headers=None*, *open_timeout=10*, *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*, ***kwds*)

Connect to the WebSocket server at `uri`.

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

`connect()` can be used as a asynchronous context manager:

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

The connection is closed automatically when exiting the context.

connect() can be used as an infinite asynchronous iterator to reconnect automatically on errors:

```
async for websocket in websockets.connect(...):
    try:
        ...
    except websockets.ConnectionClosed:
        continue
```

The connection is closed automatically after each iteration of the loop.

If an error occurs while establishing the connection, *connect()* retries with exponential backoff. The backoff delay starts at three seconds and increases up to one minute.

If an error occurs in the body of the loop, you can handle the exception and *connect()* will reconnect with the next iteration; or you can let the exception bubble up and break out of the loop. This lets you decide which errors trigger a reconnection and which errors are fatal.

Parameters

- **uri** (*str*) – URI of the WebSocket server.
- **create_protocol** (*Optional[Callable[[Any], WebSocketClientProtocol]]*) – factory for the asyncio.Protocol managing the connection; defaults to *WebSocketClientProtocol*; may be set to a wrapper or a subclass to customize connection handling.
- **logger** (*Optional[LoggerLike]*) – logger for this connection; defaults to `logging.getLogger("websockets.client")`; see the *logging guide* for details.
- **compression** (*Optional[str]*) – shortcut that enables the "permessage-deflate" extension by default; may be set to `None` to disable compression; see the *compression guide* for details.
- **origin** (*Optional[Origin]*) – value of the `Origin` header. This is useful when connecting to a server that validates the `Origin` header to defend against Cross-Site WebSocket Hijacking attacks.
- **extensions** (*Optional[Sequence[ClientExtensionFactory]]*) – list of supported extensions, in order in which they should be tried.
- **subprotocols** (*Optional[Sequence[Subprotocol]]*) – list of supported subprotocols, in order of decreasing preference.
- **extra_headers** (*Optional[HeadersLike]*) – arbitrary HTTP headers to add to the request.
- **open_timeout** (*Optional[float]*) – timeout for opening the connection in seconds; `None` to disable the timeout

See *WebSocketCommonProtocol* for the documentation of `ping_interval`, `ping_timeout`, `close_timeout`, `max_size`, `max_queue`, `read_limit`, and `write_limit`.

Any other keyword arguments are passed the event loop's `create_connection()` method.

For example:

- You can set `ssl` to a `SSLContext` to enforce TLS settings. When connecting to a `wss://` URI, if `ssl` isn't provided, a TLS context is created with `create_default_context()`.
- You can set `host` and `port` to connect to a different host and port from those found in `uri`. 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` header.

Returns
WebSocket connection.

Return type
WebSocketClientProtocol

Raises

- ***InvalidURI*** – if `uri` isn't a valid WebSocket URI.
- ***InvalidHandshake*** – if the opening handshake fails.
- **TimeoutError** – if the opening handshake times out.

`await websockets.client.`**`unix_connect`**(*path*, *uri='ws://localhost/'*, ***, *create_protocol=None*, *logger=None*, *compression='deflate'*, *origin=None*, *extensions=None*, *subprotocols=None*, *extra_headers=None*, *open_timeout=10*, *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*, ***kwds*)

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

This function builds upon 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** (*Optional[str]*) – file system path to the Unix socket.
- **uri** (*str*) – URI of the WebSocket server; the host is used in the TLS handshake for secure connections and in the `Host` header.

Using a connection

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

WebSocket client connection.

WebSocketClientProtocol provides *recv()* and *send()* coroutines for receiving and sending messages.

It supports asynchronous iteration to receive incoming messages:

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

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

See *connect()* for the documentation of `logger`, `origin`, `extensions`, `subprotocols`, and `extra_headers`.

See *WebSocketCommonProtocol* for the documentation of `ping_interval`, `ping_timeout`, `close_timeout`, `max_size`, `max_queue`, `read_limit`, and `write_limit`.

await recv()

Receive the next message.

When the connection is closed, *recv()* raises *ConnectionClosed*. Specifically, it raises *ConnectionClosedOK* after a normal connection closure and *ConnectionClosedError* after a protocol error or a network failure. This is how you detect the end of the message stream.

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()`.

Returns

A string (`str`) for a Text frame. A bytestring (`bytes`) for a Binary frame.

Return type
Data

Raises

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

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 to enable fragmentation. 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.

send() rejects dict-like objects because this is often an error. (If you want to send the keys of a dict-like object as fragments, call its `keys()` method and pass the result to *send()*.)

Canceling *send()* is discouraged. Instead, you should close the connection with *close()*. Indeed, there are only two situations where *send()* may yield control to the event loop and then get canceled; in both cases, *close()* has the same effect and is more clear:

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 that yields control. Stopping in the middle of a fragmented message will cause a protocol error and the connection will be closed.

When the connection is closed, *send()* raises *ConnectionClosed*. Specifically, it raises *ConnectionClosedOK* after a normal connection closure and *ConnectionClosedError* after a protocol error or a network failure.

Parameters
message (*Union[Data, Iterable[Data], AsyncIterable[Data]]*) – message to send.

Raises

- ***ConnectionClosed*** – when the connection is closed.
- **TypeError** – if `message` doesn't have a supported type.

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()* after *close()*.

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 take care of closing the TCP connection.

Parameters

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

await wait_closed()

Wait until the connection is closed.

This coroutine is identical to the *closed* attribute, except it can be awaited.

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

await ping(*data=None*)

Send a Ping.

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

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.

Parameters
data (*Optional[Data]*) – payload of the ping; a string will be encoded to UTF-8; or `None` to generate a payload containing four random bytes.

Returns

A future that will be completed when the corresponding pong is received. You can ignore it if you don't intend to wait.

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

Return type
Future

Raises

- *ConnectionClosed* – when the connection is closed.
- RuntimeError – if another ping was sent with the same data and the corresponding pong wasn't received yet.

`await pong(data=b'')`

Send a Pong.

An unsolicited pong may serve as a unidirectional heartbeat.

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

Parameters

data (Data) – payload of the pong; a string will be encoded to UTF-8.

Raises

ConnectionClosed – when the connection is closed.

WebSocket connection objects also provide these attributes:

`id: uuid.UUID`

Unique identifier of the connection. Useful in logs.

`logger: LoggerLike`

Logger for this connection.

`property local_address: Any`

Local address of the connection.

For IPv4 connections, this is a `(host, port)` tuple.

The format of the address depends on the address family; see `getsockname()`.

`None` if the TCP connection isn't established yet.

`property remote_address: Any`

Remote address of the connection.

For IPv4 connections, this is a `(host, port)` tuple.

The format of the address depends on the address family; see `getpeername()`.

`None` if the TCP connection isn't established yet.

`property open: bool`

`True` when the connection is open; `False` otherwise.

This attribute may be used to detect disconnections. However, this approach is discouraged per the EAFP principle. Instead, you should handle *ConnectionClosed* exceptions.

`property closed: bool`

`True` when the connection is closed; `False` otherwise.

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

The following attributes are available after the opening handshake, once the WebSocket connection is open:

`path: str`

Path of the opening handshake request.

request_headers: *Headers*

Opening handshake request headers.

response_headers: *Headers*

Opening handshake response headers.

subprotocol: Optional[*Subprotocol*]

Subprotocol, if one was negotiated.

The following attributes are available after the closing handshake, once the WebSocket connection is closed:

property close_code: Optional[int]

WebSocket close code, defined in section 7.1.5 of RFC 6455.

None if the connection isn't closed yet.

property close_reason: Optional[str]

WebSocket close reason, defined in section 7.1.6 of RFC 6455.

None if the connection isn't closed yet.

3.1.2 Sans-I/O

class websockets.client.**ClientConnection**(*wsuri*, *origin=None*, *extensions=None*, *subprotocols=None*, *state=State.CONNECTING*, *max_size=2**20*, *logger=None*)

Sans-I/O implementation of a WebSocket client connection.

Parameters

- **wsuri** (WebSocketURI) – URI of the WebSocket server, parsed with *parse_uri()*.
- **origin** (*Optional[Origin]*) – value of the Origin header. This is useful when connecting to a server that validates the Origin header to defend against Cross-Site WebSocket Hijacking attacks.
- **extensions** (*Optional[Sequence[ClientExtensionFactory]]*) – list of supported extensions, in order in which they should be tried.
- **subprotocols** (*Optional[Sequence[Subprotocol]]*) – list of supported subprotocols, in order of decreasing preference.
- **state** (State) – initial state of the WebSocket connection.
- **max_size** (*Optional[int]*) – maximum size of incoming messages in bytes; None to disable the limit.
- **logger** (*Optional[LoggerLike]*) – logger for this connection; defaults to logging.getLogger("websockets.client"); see the *logging guide* for details.

receive_data(*data*)

Receive data from the network.

After calling this method:

- You must call *data_to_send()* and send this data to the network.
- You should call *events_received()* and process resulting events.

Raises

EOFError – if *receive_eof()* was called earlier.

`receive_eof()`

Receive the end of the data stream from the network.

After calling this method:

- You must call *data_to_send()* and send this data to the network.
- You aren't expected to call *events_received()*; it won't return any new events.

Raises
EOFError – if *receive_eof()* was called earlier.

`connect()`

Create a handshake request to open a connection.

You must send the handshake request with *send_request()*.

You can modify it before sending it, for example to add HTTP headers.

Returns
WebSocket handshake request event to send to the server.

Return type
Request

`send_request(request)`

Send a handshake request to the server.

Parameters
request (Request) – WebSocket handshake request event.

`send_continuation(data, fin)`

Send a Continuation frame.

Parameters

- **data** (*bytes*) – payload containing the same kind of data as the initial frame.
- **fin** (*bool*) – FIN bit; set it to True if this is the last frame of a fragmented message and to False otherwise.

Raises
ProtocolError – if a fragmented message isn't in progress.

`send_text(data, fin=True)`

Send a Text frame.

Parameters

- **data** (*bytes*) – payload containing text encoded with UTF-8.
- **fin** (*bool*) – FIN bit; set it to False if this is the first frame of a fragmented message.

Raises
ProtocolError – if a fragmented message is in progress.

`send_binary(data, fin=True)`

Send a Binary frame.

Parameters

- **data** (*bytes*) – payload containing arbitrary binary data.
- **fin** (*bool*) – FIN bit; set it to False if this is the first frame of a fragmented message.

Raises
ProtocolError – if a fragmented message is in progress.

`send_close`(*code=None*, *reason=''*)

Send a Close frame.

Parameters

- **code** (*Optional[int]*) – close code.
- **reason** (*str*) – close reason.

Raises
ProtocolError – if a fragmented message is being sent, if the code isn't valid, or if a reason is provided without a code

`send_ping`(*data*)

Send a Ping frame.

Parameters
data (*bytes*) – payload containing arbitrary binary data.

`send_pong`(*data*)

Send a Pong frame.

Parameters
data (*bytes*) – payload containing arbitrary binary data.

`fail`(*code*, *reason=''*)

Fail the WebSocket connection.

Parameters

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

Raises
ProtocolError – if the code isn't valid.

`events_received`()

Fetch events generated from data received from the network.

Call this method immediately after any of the `receive_*()` methods.

Process resulting events, likely by passing them to the application.

Returns
Events read from the connection.

Return type
List[*Event*]

`data_to_send`()

Obtain data to send to the network.

Call this method immediately after any of the `receive_*()`, `send_*()`, or *fail()* methods.

Write resulting data to the connection.

The empty bytestring *SEND_EOF* signals the end of the data stream. When you receive it, half-close the TCP connection.

Returns
Data to write to the connection.

Return type
List[bytes]

`close_expected()`

Tell if the TCP connection is expected to close soon.

Call this method immediately after any of the `receive_*()` or *fail()* methods.

If it returns `True`, schedule closing the TCP connection after a short timeout if the other side hasn't already closed it.

Returns
Whether the TCP connection is expected to close soon.

Return type
bool

`id: uuid.UUID`

Unique identifier of the connection. Useful in logs.

`logger: LoggerLike`

Logger for this connection.

`property state: State`

WebSocket connection state.

Defined in 4.1, 4.2, 7.1.3, and 7.1.4 of **RFC 6455**.

`property close_code: Optional[int]`

WebSocket close code.

`None` if the connection isn't closed yet.

`property close_reason: Optional[str]`

WebSocket close reason.

`None` if the connection isn't closed yet.

`property close_exc: ConnectionClosed`

Exception to raise when trying to interact with a closed connection.

Don't raise this exception while the connection *state* is *CLOSING*; wait until it's *CLOSED*.

Indeed, the exception includes the close code and reason, which are known only once the connection is closed.

Raises
`AssertionError` – if the connection isn't closed yet.

3.2 Server

3.2.1 asyncio

Starting a server

await websockets.server.**serve**(*ws_handler*, *host=None*, *port=None*, ***, *create_protocol=None*, *logger=None*, *compression='deflate'*, *origins=None*, *extensions=None*, *subprotocols=None*, *extra_headers=None*, *process_request=None*, *select_subprotocol=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*, ***kwds*)

Start a WebSocket server listening on `host` and `port`.

Whenever a client connects, the server creates a *WebSocketServerProtocol*, performs the opening handshake, and delegates to the connection handler, `ws_handler`.

The handler receives the *WebSocketServerProtocol* and uses it to send and receive messages.

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 object provides *close()* and *wait_closed()* methods for shutting down the server.

serve() can be used as an asynchronous context manager:

```
stop = asyncio.Future()  # set this future to exit the server

async with serve(...):
    await stop
```

The server is shut down automatically when exiting the context.

Parameters

- **ws_handler** (*Callable[[WebSocketServerProtocol, str], Awaitable[Any]]*) – connection handler. It must be a coroutine accepting two arguments: the WebSocket connection, which is a *WebSocketServerProtocol*, and the path of the request.
- **host** (*Optional[Union[str, Sequence[str]]]*) – network interfaces the server is bound to; see `create_server()` for details.
- **port** (*Optional[int]*) – TCP port the server listens on; see `create_server()` for details.
- **create_protocol** (*Optional[Callable[[Any], WebSocketServerProtocol]]*) – factory for the `asyncio.Protocol` managing the connection; defaults to *WebSocketServerProtocol*; may be set to a wrapper or a subclass to customize connection handling.
- **logger** (*Optional[LoggerLike]*) – logger for this server; defaults to `logging.getLogger("websockets.server")`; see the *logging guide* for details.
- **compression** (*Optional[str]*) – shortcut that enables the "permessage-deflate" extension by default; may be set to `None` to disable compression; see the *compression guide* for details.
- **origins** (*Optional[Sequence[Optional[Origin]]]*) – acceptable values of the `Origin` header; include `None` in the list if the lack of an origin is acceptable. This is useful for defending against Cross-Site WebSocket Hijacking attacks.
- **extensions** (*Optional[Sequence[ServerExtensionFactory]]*) – list of supported extensions, in order in which they should be tried.
- **subprotocols** (*Optional[Sequence[Subprotocol]]*) – list of supported subprotocols, in order of decreasing preference.

- **extra_headers** (*Union[HeadersLike, Callable[[str, Headers], HeadersLike]]*) – arbitrary HTTP headers to add to the request; this can be a *HeadersLike* or a callable taking the request path and headers in arguments and returning a *HeadersLike*.
- **process_request** (*Optional[Callable[[str, Headers], Awaitable[Optional[Tuple[http.HTTPStatus, HeadersLike, bytes]]]]]*) – intercept HTTP request before the opening handshake; see *process_request()* for details.
- **select_subprotocol** (*Optional[Callable[[Sequence[Subprotocol], Sequence[Subprotocol]], Subprotocol]]*) – select a subprotocol supported by the client; see *select_subprotocol()* for details.

See *WebSocketCommonProtocol* for the documentation of `ping_interval`, `ping_timeout`, `close_timeout`, `max_size`, `max_queue`, `read_limit`, and `write_limit`.

Any other keyword arguments are passed the event loop's `create_server()` method.

For example:

- You can set `ssl` to a `SSLContext` to enable TLS.
- You can set `sock` to a `socket` that you created outside of websockets.

Returns
WebSocket server.

Return type
WebSocketServer

`await websockets.server.unix_serve(`*ws_handler*, *path=None*, ***, *create_protocol=None*, *logger=None*, *compression='deflate'*, *origins=None*, *extensions=None*, *subprotocols=None*, *extra_headers=None*, *process_request=None*, *select_subprotocol=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*, ***kwds*`)`

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

This function builds upon 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 (*Optional[str]*) – file system path to the Unix socket.

Stopping a server

`class websockets.server.WebSocketServer(`*logger=None*`)`

WebSocket server returned by *serve()*.

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

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

Parameters
logger (*Optional[LoggerLike]*) – logger for this server; defaults to `logging.getLogger("websockets.server")`; see the *logging guide* for details.

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.

await wait_closed()

Wait until the server is closed.

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

To ensure a fast shutdown, a connection handler should always be awaiting at least one of:

- *recv()*: when the connection is closed, it raises *ConnectionClosedOK*;
- *wait_closed()*: when the connection is closed, it returns.

Then the connection handler is immediately notified of the shutdown; it can clean up and exit.

sockets

List of socket objects the server is listening on.

None if the server is closed.

Using a connection

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

WebSocket server connection.

WebSocketServerProtocol provides *recv()* and *send()* coroutines for receiving and sending messages.

It supports asynchronous iteration to receive messages:

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

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

You may customize the opening handshake in a subclass by overriding *process_request()* or *select_subprotocol()*.

Parameters
ws_server (WebSocketServer) – WebSocket server that created this connection.

See *serve()* for the documentation of `ws_handler`, `logger`, `origins`, `extensions`, `subprotocols`, and `extra_headers`.

See *WebSocketCommonProtocol* for the documentation of `ping_interval`, `ping_timeout`, `close_timeout`, `max_size`, `max_queue`, `read_limit`, and `write_limit`.

await recv()

Receive the next message.

When the connection is closed, *recv()* raises *ConnectionClosed*. Specifically, it raises *ConnectionClosedOK* after a normal connection closure and *ConnectionClosedError* after a protocol error or a network failure. This is how you detect the end of the message stream.

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()`.

Returns

A string (`str`) for a Text frame. A bytestring (`bytes`) for a Binary frame.

Return type
Data

Raises

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

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 to enable fragmentation. 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.

send() rejects dict-like objects because this is often an error. (If you want to send the keys of a dict-like object as fragments, call its `keys()` method and pass the result to *send()*.)

Canceling *send()* is discouraged. Instead, you should close the connection with *close()*. Indeed, there are only two situations where *send()* may yield control to the event loop and then get canceled; in both cases, *close()* has the same effect and is more clear:

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 that yields control. Stopping in the middle of a fragmented message will cause a protocol error and the connection will be closed.

When the connection is closed, *send()* raises *ConnectionClosed*. Specifically, it raises *ConnectionClosedOK* after a normal connection closure and *ConnectionClosedError* after a protocol error or a network failure.

Parameters
message (*Union[Data, Iterable[Data], AsyncIterable[Data]]*) – message to send.

Raises

- *ConnectionClosed* – when the connection is closed.
- **TypeError** – if `message` doesn't have a supported type.

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()* after *close()*.

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 take care of closing the TCP connection.

Parameters

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

await wait_closed()

Wait until the connection is closed.

This coroutine is identical to the *closed* attribute, except it can be awaited.

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

await ping(*data=None*)

Send a Ping.

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

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.

Parameters

data (*Optional[Data]*) – payload of the ping; a string will be encoded to UTF-8; or `None` to generate a payload containing four random bytes.

Returns

A future that will be completed when the corresponding pong is received. You can ignore it if you don't intend to wait.

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

Return type

Future

Raises

- *ConnectionClosed* – when the connection is closed.

- **RuntimeError** – if another ping was sent with the same data and the corresponding pong wasn't received yet.

await pong(*data=b''*)

Send a Pong.

An unsolicited pong may serve as a unidirectional heartbeat.

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

Parameters
data (Data) – payload of the pong; a string will be encoded to UTF-8.

Raises
ConnectionClosed – when the connection is closed.

You can customize the opening handshake in a subclass by overriding these methods:

await process_request(*path*, *request_headers*)

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

You may override this method 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.

You may also override this method with the `process_request` argument of *serve()* and *WebSocketServerProtocol*. 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.

Returns

None to continue the WebSocket handshake normally.

An HTTP response, represented by a 3-uple of the response status, headers, and body, to abort the WebSocket handshake and return that HTTP response instead.

Return type
Optional[Tuple[http.HTTPStatus, *HeadersLike*, 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 subprotocol 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. Many servers providing a subprotocol will require that the client uses that subprotocol. Such rules can be implemented in a subclass.

You may also override this method with the `select_subprotocol` argument of *serve()* and *WebSocketServerProtocol*.

Parameters

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

Returns

Selected subprotocol.

`None` to continue without a subprotocol.

Return type
Optional[*Subprotocol*]

WebSocket connection objects also provide these attributes:

id: *uuid.UUID*

Unique identifier of the connection. Useful in logs.

logger: *LoggerLike*

Logger for this connection.

property local_address: *Any*

Local address of the connection.

For IPv4 connections, this is a `(host, port)` tuple.

The format of the address depends on the address family; see `getsockname()`.

`None` if the TCP connection isn't established yet.

property remote_address: *Any*

Remote address of the connection.

For IPv4 connections, this is a `(host, port)` tuple.

The format of the address depends on the address family; see `getpeername()`.

`None` if the TCP connection isn't established yet.

property open: *bool*

`True` when the connection is open; `False` otherwise.

This attribute may be used to detect disconnections. However, this approach is discouraged per the EAFP principle. Instead, you should handle *ConnectionClosed* exceptions.

property closed: *bool*

`True` when the connection is closed; `False` otherwise.

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

The following attributes are available after the opening handshake, once the WebSocket connection is open:

path: *str*

Path of the opening handshake request.

request_headers: *Headers*

Opening handshake request headers.

response_headers: *Headers*

Opening handshake response headers.

subprotocol: Optional[*Subprotocol*]

Subprotocol, if one was negotiated.

The following attributes are available after the closing handshake, once the WebSocket connection is closed:

property close_code: Optional[int]

WebSocket close code, defined in section 7.1.5 of RFC 6455.

None if the connection isn't closed yet.

property close_reason: Optional[str]

WebSocket close reason, defined in section 7.1.6 of RFC 6455.

None if the connection isn't closed yet.

Basic authentication

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

websockets.auth.**basic_auth_protocol_factory**(*realm=None*, *credentials=None*, *check_credentials=None*, *create_protocol=None*)

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"),
    )
)
```

Parameters

- **realm** (*Optional[str]*) – 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** (*Optional[Union[Tuple[str, str], Iterable[Tuple[str, str]]]]*) – defines hard coded authorized credentials. It can be a (username, password) pair or a list of such pairs.
- **check_credentials** (*Optional[Callable[[str, str], Awaitable[bool]]]*) – defines a coroutine that verifies credentials. This coroutine receives username and password arguments and returns a bool. One of credentials or check_credentials must be provided but not both.
- **create_protocol** (*Optional[Callable[[Any], BasicAuthWebSocketServerProtocol]]*) – factory that creates the protocol. By default, this is *BasicAuthWebSocketServerProtocol*. It can be replaced by a subclass.

Raises

TypeError – if the credentials or check_credentials argument is wrong.

class websockets.auth.**BasicAuthWebSocketServerProtocol**(**args*, *realm=None*, *check_credentials=None*, ***kwargs*)

WebSocket server protocol that enforces HTTP Basic Auth.

realm: str = ''

Scope of protection.

If provided, it should contain only ASCII characters because the encoding of non-ASCII characters is undefined.

username: Optional[str] = None

Username of the authenticated user.

await **check_credentials**(*username*, *password*)

Check whether credentials are authorized.

This coroutine may be overridden in a subclass, for example to authenticate against a database or an external service.

Parameters

- **username** (*str*) – HTTP Basic Auth username.
- **password** (*str*) – HTTP Basic Auth password.

Returns
True if the handshake should continue; False if it should fail with a HTTP 401 error.

Return type
bool

3.2.2 Sans-I/O

class websockets.server.**ServerConnection**(*origins=None*, *extensions=None*, *subprotocols=None*, *state=State.CONNECTING*, *max_size=2**20*, *logger=None*)

Sans-I/O implementation of a WebSocket server connection.

Parameters

- **origins** (*Optional[Sequence[Optional[Origin]]]*) – acceptable values of the Origin header; include None in the list if the lack of an origin is acceptable. This is useful for defending against Cross-Site WebSocket Hijacking attacks.
- **extensions** (*List[Extension]*) – list of supported extensions, in order in which they should be tried.
- **subprotocols** (*Optional[Sequence[Subprotocol]]*) – list of supported subprotocols, in order of decreasing preference.
- **state** (State) – initial state of the WebSocket connection.
- **max_size** (*Optional[int]*) – maximum size of incoming messages in bytes; None to disable the limit.
- **logger** (*Union[Logger, LoggerAdapter]*) – logger for this connection; defaults to logging.getLogger("websockets.client"); see the *logging guide* for details.

`receive_data`(*data*)

Receive data from the network.

After calling this method:

- You must call *data_to_send()* and send this data to the network.
- You should call *events_received()* and process resulting events.

Raises
EOFError – if *receive_eof()* was called earlier.

`receive_eof`()

Receive the end of the data stream from the network.

After calling this method:

- You must call *data_to_send()* and send this data to the network.
- You aren't expected to call *events_received()*; it won't return any new events.

Raises
EOFError – if *receive_eof()* was called earlier.

`accept`(*request*)

Create a handshake response to accept the connection.

If the connection cannot be established, the handshake response actually rejects the handshake.

You must send the handshake response with *send_response()*.

You can modify it before sending it, for example to add HTTP headers.

Parameters
request (Request) – WebSocket handshake request event received from the client.

Returns
WebSocket handshake response event to send to the client.

Return type
Response

`reject`(*status*, *text*)

Create a handshake response to reject the connection.

A short plain text response is the best fallback when failing to establish a WebSocket connection.

You must send the handshake response with *send_response()*.

You can modify it before sending it, for example to alter HTTP headers.

Parameters

- **status** (*HTTPStatus*) – HTTP status code.
- **text** (*str*) – HTTP response body; will be encoded to UTF-8.

Returns
WebSocket handshake response event to send to the client.

Return type
Response

`send_response`(*response*)

Send a handshake response to the client.

Parameters
response (Response) – WebSocket handshake response event to send.

`send_continuation`(*data*, *fin*)

Send a Continuation frame.

Parameters

- **data** (*bytes*) – payload containing the same kind of data as the initial frame.
- **fin** (*bool*) – FIN bit; set it to True if this is the last frame of a fragmented message and to False otherwise.

Raises
ProtocolError – if a fragmented message isn't in progress.

`send_text`(*data*, *fin=True*)

Send a Text frame.

Parameters

- **data** (*bytes*) – payload containing text encoded with UTF-8.
- **fin** (*bool*) – FIN bit; set it to False if this is the first frame of a fragmented message.

Raises
ProtocolError – if a fragmented message is in progress.

`send_binary`(*data*, *fin=True*)

Send a Binary frame.

Parameters

- **data** (*bytes*) – payload containing arbitrary binary data.
- **fin** (*bool*) – FIN bit; set it to False if this is the first frame of a fragmented message.

Raises
ProtocolError – if a fragmented message is in progress.

`send_close`(*code=None*, *reason=''*)

Send a Close frame.

Parameters

- **code** (*Optional[int]*) – close code.
- **reason** (*str*) – close reason.

Raises
ProtocolError – if a fragmented message is being sent, if the code isn't valid, or if a reason is provided without a code

`send_ping`(*data*)

Send a Ping frame.

Parameters
data (*bytes*) – payload containing arbitrary binary data.

`send_pong`(*data*)

Send a Pong frame.

Parameters
data (*bytes*) – payload containing arbitrary binary data.

`fail`(*code*, *reason=''*)

Fail the WebSocket connection.

Parameters

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

Raises
ProtocolError – if the code isn't valid.

`events_received`()

Fetch events generated from data received from the network.

Call this method immediately after any of the `receive_*()` methods.

Process resulting events, likely by passing them to the application.

Returns
Events read from the connection.

Return type
List[*Event*]

`data_to_send`()

Obtain data to send to the network.

Call this method immediately after any of the `receive_*()`, `send_*()`, or *fail()* methods.

Write resulting data to the connection.

The empty bytestring *SEND_EOF* signals the end of the data stream. When you receive it, half-close the TCP connection.

Returns
Data to write to the connection.

Return type
List[bytes]

`close_expected`()

Tell if the TCP connection is expected to close soon.

Call this method immediately after any of the `receive_*()` or *fail()* methods.

If it returns `True`, schedule closing the TCP connection after a short timeout if the other side hasn't already closed it.

Returns
Whether the TCP connection is expected to close soon.

Return type
bool

`id: uuid.UUID`

Unique identifier of the connection. Useful in logs.

logger: *LoggerLike*

Logger for this connection.

property state: *State*

WebSocket connection state.

Defined in 4.1, 4.2, 7.1.3, and 7.1.4 of **RFC 6455**.

property close_code: Optional[int]

WebSocket close code.

None if the connection isn't closed yet.

property close_reason: Optional[str]

WebSocket close reason.

None if the connection isn't closed yet.

property close_exc: *ConnectionClosed*

Exception to raise when trying to interact with a closed connection.

Don't raise this exception while the connection *state* is *CLOSING*; wait until it's *CLOSED*.

Indeed, the exception includes the close code and reason, which are known only once the connection is closed.

Raises

AssertionError – if the connection isn't closed yet.

3.3 Both sides

3.3.1 asyncio

class websockets.legacy.protocol.**WebSocketCommonProtocol**(**, logger=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*)

WebSocket connection.

WebSocketCommonProtocol provides APIs shared between WebSocket servers and clients. You shouldn't use it directly. Instead, use *WebSocketClientProtocol* or *WebSocketServerProtocol*.

This documentation focuses on low-level details that aren't covered in the documentation of *WebSocketClientProtocol* and *WebSocketServerProtocol* for the sake of simplicity.

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 for completing the closing handshake and terminating the TCP connection. For legacy reasons, *close()* completes in at most `5 * close_timeout` seconds for clients and `4 * close_timeout` for servers.

See the discussion of *timeouts* for details.

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

- on the client side, when *`connect()`* is used as a context manager;
- on the server side, when the connection handler terminates;

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. If a larger message 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`. 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`).

See the discussion of *memory usage* for details.

Parameters

- **`logger`** (*`Optional[LoggerLike]`*) – logger for this connection; defaults to `logging.getLogger("websockets.protocol")`; see the *logging guide* for details.
- **`ping_interval`** (*`Optional[float]`*) – delay between keepalive pings in seconds; `None` to disable keepalive pings.
- **`ping_timeout`** (*`Optional[float]`*) – timeout for keepalive pings in seconds; `None` to disable timeouts.
- **`close_timeout`** (*`Optional[float]`*) – timeout for closing the connection in seconds; for legacy reasons, the actual timeout is 4 or 5 times larger.
- **`max_size`** (*`Optional[int]`*) – maximum size of incoming messages in bytes; `None` to disable the limit.
- **`max_queue`** (*`Optional[int]`*) – maximum number of incoming messages in receive buffer; `None` to disable the limit.
- **`read_limit`** (*`int`*) – high-water mark of read buffer in bytes.
- **`write_limit`** (*`int`*) – high-water mark of write buffer in bytes.

`await recv()`

Receive the next message.

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

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().

Returns

A string (str) for a Text frame. A bytestring (bytes) for a Binary frame.

Return type
Data

Raises

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

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 to enable fragmentation. 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.

send() rejects dict-like objects because this is often an error. (If you want to send the keys of a dict-like object as fragments, call its keys() method and pass the result to *send()*.)

Canceling *send()* is discouraged. Instead, you should close the connection with *close()*. Indeed, there are only two situations where *send()* may yield control to the event loop and then get canceled; in both cases, *close()* has the same effect and is more clear:

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 that yields control. Stopping in the middle of a fragmented message will cause a protocol error and the connection will be closed.

When the connection is closed, *send()* raises *ConnectionClosed*. Specifically, it raises *ConnectionClosedOK* after a normal connection closure and *ConnectionClosedError* after a protocol error or a network failure.

Parameters
message(*Union[Data, Iterable[Data], AsyncIterable[Data]]*) – message to send.

Raises

- **ConnectionClosed** – when the connection is closed.
- **TypeError** – if message doesn't have a supported type.

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()* after *close()*.

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 take care of closing the TCP connection.

Parameters

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

await wait_closed()

Wait until the connection is closed.

This coroutine is identical to the *closed* attribute, except it can be awaited.

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

await ping(*data=None*)

Send a Ping.

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

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.

Parameters

data (*Optional[Data]*) – payload of the ping; a string will be encoded to UTF-8; or `None` to generate a payload containing four random bytes.

Returns

A future that will be completed when the corresponding pong is received. You can ignore it if you don't intend to wait.

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

Return type

Future

Raises

- ***ConnectionClosed*** – when the connection is closed.
- **RuntimeError** – if another ping was sent with the same data and the corresponding pong wasn't received yet.

await pong(*data=b''*)

Send a Pong.

An unsolicited pong may serve as a unidirectional heartbeat.

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

Parameters

data (*Data*) – payload of the pong; a string will be encoded to UTF-8.

Raises
ConnectionClosed – when the connection is closed.

WebSocket connection objects also provide these attributes:

id: uuid.UUID

Unique identifier of the connection. Useful in logs.

logger: *LoggerLike*

Logger for this connection.

property local_address: Any

Local address of the connection.

For IPv4 connections, this is a `(host, port)` tuple.

The format of the address depends on the address family; see `getsockname()`.

`None` if the TCP connection isn't established yet.

property remote_address: Any

Remote address of the connection.

For IPv4 connections, this is a `(host, port)` tuple.

The format of the address depends on the address family; see `getpeername()`.

`None` if the TCP connection isn't established yet.

property open: bool

`True` when the connection is open; `False` otherwise.

This attribute may be used to detect disconnections. However, this approach is discouraged per the EAFP principle. Instead, you should handle *ConnectionClosed* exceptions.

property closed: bool

`True` when the connection is closed; `False` otherwise.

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

The following attributes are available after the opening handshake, once the WebSocket connection is open:

path: str

Path of the opening handshake request.

request_headers: *Headers*

Opening handshake request headers.

response_headers: *Headers*

Opening handshake response headers.

subprotocol: Optional[*Subprotocol*]

Subprotocol, if one was negotiated.

The following attributes are available after the closing handshake, once the WebSocket connection is closed:

property close_code: Optional[int]

WebSocket close code, defined in section 7.1.5 of RFC 6455.

`None` if the connection isn't closed yet.

property close_reason: Optional[str]

WebSocket close reason, defined in section 7.1.6 of RFC 6455.

None if the connection isn't closed yet.

3.3.2 Sans-I/O

class websockets.connection.Connection(*side*, *state=State.OPEN*, *max_size=2**20*, *logger=None*)

Sans-I/O implementation of a WebSocket connection.

Parameters

- **side** (Side) – *CLIENT* or *SERVER*.
- **state** (State) – initial state of the WebSocket connection.
- **max_size** (*Optional[int]*) – maximum size of incoming messages in bytes; None to disable the limit.
- **logger** (*Optional[LoggerLike]*) – logger for this connection; depending on side, defaults to logging.getLogger("websockets.client") or logging.getLogger("websockets.server"); see the *logging guide* for details.

receive_data(*data*)

Receive data from the network.

After calling this method:

- You must call *data_to_send()* and send this data to the network.
- You should call *events_received()* and process resulting events.

Raises

EOFError – if *receive_eof()* was called earlier.

receive_eof()

Receive the end of the data stream from the network.

After calling this method:

- You must call *data_to_send()* and send this data to the network.
- You aren't expected to call *events_received()*; it won't return any new events.

Raises

EOFError – if *receive_eof()* was called earlier.

send_continuation(*data*, *fin*)

Send a Continuation frame.

Parameters

- **data** (*bytes*) – payload containing the same kind of data as the initial frame.
- **fin** (*bool*) – FIN bit; set it to True if this is the last frame of a fragmented message and to False otherwise.

Raises

ProtocolError – if a fragmented message isn't in progress.

send_text(*data*, *fin=True*)

Send a Text frame.

Parameters

- **data** (*bytes*) – payload containing text encoded with UTF-8.
- **fin** (*bool*) – FIN bit; set it to `False` if this is the first frame of a fragmented message.

Raises

ProtocolError – if a fragmented message is in progress.

send_binary(*data*, *fin=True*)

Send a Binary frame.

Parameters

- **data** (*bytes*) – payload containing arbitrary binary data.
- **fin** (*bool*) – FIN bit; set it to `False` if this is the first frame of a fragmented message.

Raises

ProtocolError – if a fragmented message is in progress.

send_close(*code=None*, *reason=''*)

Send a Close frame.

Parameters

- **code** (*Optional[int]*) – close code.
- **reason** (*str*) – close reason.

Raises

ProtocolError – if a fragmented message is being sent, if the code isn't valid, or if a reason is provided without a code

send_ping(*data*)

Send a Ping frame.

Parameters

data (*bytes*) – payload containing arbitrary binary data.

send_pong(*data*)

Send a Pong frame.

Parameters

data (*bytes*) – payload containing arbitrary binary data.

fail(*code*, *reason=''*)

Fail the WebSocket connection.

Parameters

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

Raises

ProtocolError – if the code isn't valid.

`events_received()`

Fetch events generated from data received from the network.

Call this method immediately after any of the `receive_*()` methods.

Process resulting events, likely by passing them to the application.

Returns
Events read from the connection.

Return type
List[*Event*]

`data_to_send()`

Obtain data to send to the network.

Call this method immediately after any of the `receive_*()`, `send_*()`, or *fail()* methods.

Write resulting data to the connection.

The empty bytestring *SEND_EOF* signals the end of the data stream. When you receive it, half-close the TCP connection.

Returns
Data to write to the connection.

Return type
List[bytes]

`close_expected()`

Tell if the TCP connection is expected to close soon.

Call this method immediately after any of the `receive_*()` or *fail()* methods.

If it returns `True`, schedule closing the TCP connection after a short timeout if the other side hasn't already closed it.

Returns
Whether the TCP connection is expected to close soon.

Return type
bool

`id: UUID`

Unique identifier of the connection. Useful in logs.

`logger: Union[Logger, LoggerAdapter]`

Logger for this connection.

`property state: State`

WebSocket connection state.

Defined in 4.1, 4.2, 7.1.3, and 7.1.4 of **RFC 6455**.

`property close_code: Optional[int]`

WebSocket close code.

`None` if the connection isn't closed yet.

`property close_reason: Optional[str]`

WebSocket close reason.

`None` if the connection isn't closed yet.

property close_exc: *ConnectionClosed*

Exception to raise when trying to interact with a closed connection.

Don't raise this exception while the connection *state* is *CLOSING*; wait until it's *CLOSED*.

Indeed, the exception includes the close code and reason, which are known only once the connection is closed.

Raises

AssertionError – if the connection isn't closed yet.

class websockets.connection.**Side**(*value*)

A WebSocket connection is either a server or a client.

SERVER = 0

CLIENT = 1

class websockets.connection.**State**(*value*)

A WebSocket connection is in one of these four states.

CONNECTING = 0

OPEN = 1

CLOSING = 2

CLOSED = 3

websockets.connection.**SEND_EOF = b''**

Sentinel signaling that the TCP connection must be half-closed.

3.4 Utilities

3.4.1 Broadcast

websockets.**broadcast**(*websockets*, *message*)

Broadcast a message to several WebSocket connections.

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.

broadcast() pushes the message synchronously to all connections even if their write buffers are overflowing. There's no backpressure.

broadcast() skips silently connections that aren't open in order to avoid errors on connections where the closing handshake is in progress.

If you broadcast messages faster than a connection can handle them, messages will pile up in its write buffer until the connection times out. Keep low values for `ping_interval` and `ping_timeout` to prevent excessive memory usage by slow connections when you use *broadcast()*.

Unlike *send()*, *broadcast()* doesn't support sending fragmented messages. Indeed, fragmentation is useful for sending large messages without buffering them in memory, while *broadcast()* buffers one copy per connection as fast as possible.

Parameters

- **websockets** (*Iterable[WebSocketCommonProtocol]*) – WebSocket connections to which the message will be sent.
- **message** (Data) – message to send.

Raises

- **RuntimeError** – if a connection is busy sending a fragmented message.
- **TypeError** – if `message` doesn't have a supported type.

3.4.2 WebSocket events

class `websockets.frames.`**Frame**(*opcode*, *data*, *fin=True*, *rsv1=False*, *rsv2=False*, *rsv3=False*)

WebSocket frame.

opcode

Opcode.

Type
websockets.frames.Opcode

data

Payload data.

Type
bytes

fin

FIN bit.

Type
bool

rsv1

RSV1 bit.

Type
bool

rsv2

RSV2 bit.

Type
bool

rsv3

RSV3 bit.

Type
bool

Only these fields are needed. The MASK bit, payload length and masking-key are handled on the fly when parsing and serializing frames.

class `websockets.frames.`**Opcode**(*value*)

Opcode values for WebSocket frames.

CONT = 0

TEXT = 1

BINARY = 2

CLOSE = 8

PING = 9

PONG = 10

class websockets.frames.**Close**(*code*, *reason*)

Code and reason for WebSocket close frames.

code

Close code.

Type
int

reason

Close reason.

Type
str

3.4.3 HTTP events

class websockets.http11.**Request**(*path*, *headers*, *exception=None*)

WebSocket handshake request.

path

Request path, including optional query.

Type
str

headers

Request headers.

Type
websockets.datastructures.Headers

exception

If processing the response triggers an exception, the exception is stored in this attribute.

Type
Optional[Exception]

class websockets.http11.**Response**(*status_code*, *reason_phrase*, *headers*, *body=None*, *exception=None*)

WebSocket handshake response.

status_code

Response code.

Type
int

reason_phrase

Response reason.

Type
str

headers

Response headers.

Type
websockets.datastructures.Headers

body

Response body, if any.

Type
Optional[bytes]

exception

if processing the response triggers an exception, the exception is stored in this attribute.

Type
Optional[Exception]

class websockets.datastructures.**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 manipulating multiple values explicitly:

- *get_all()* returns a list of all values for a header;

- *raw_items()* returns an iterator of `(name, values)` pairs.

get_all(*key*)

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

Parameters
key (*str*) – header name.

raw_items()

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

exception websockets.datastructures.**MultipleValuesError**

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

3.4.4 URIs

websockets.uri.**parse_uri**(*uri*)

Parse and validate a WebSocket URI.

Parameters
uri (*str*) – WebSocket URI.

Returns
Parsed WebSocket URI.

Return type
WebSocketURI

Raises
InvalidURI – if `uri` isn't a valid WebSocket URI.

class websockets.uri.**WebSocketURI**(*secure*, *host*, *port*, *path*, *query*, *username*, *password*)

WebSocket URI.

secure

`True` for a `wss` URI, `False` for a `ws` URI.

Type
bool

host

Normalized to lower case.

Type
str

port

Always set even if it's the default.

Type
int

path

May be empty.

Type
str

query

May be empty if the URI doesn't include a query component.

Type
str

username

Available when the URI contains User Information.

Type
Optional[str]

password

Available when the URI contains User Information.

Type
Optional[str]

3.5 Exceptions

websockets.exceptions defines the following exception hierarchy:

- ***WebSocketException***
  - ***ConnectionClosed***
    - *ConnectionClosedError*
    - *ConnectionClosedOK*
  - ***InvalidHandshake***
    - *SecurityError*
    - *InvalidMessage*
    - ***InvalidHeader***
      - *InvalidHeaderFormat*
      - *InvalidHeaderValue*
      - *InvalidOrigin*
      - *InvalidUpgrade*
    - *InvalidStatus*
    - *InvalidStatusCode* (legacy)
    - ***NegotiationError***
      - *DuplicateParameter*
      - *InvalidParameterName*
      - *InvalidParameterValue*
    - *AbortHandshake*
    - *RedirectHandshake*
  - *InvalidState*
  - *InvalidURI*

- *PayloadTooBig*
- *ProtocolError*

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()*.

status

HTTP status code.

Type
HTTPStatus

headers

HTTP response headers.

Type
Headers

body

HTTP response body.

Type
bytes

exception websockets.exceptions.**ConnectionClosed**(*rcvd*, *sent*, *rcvd_then_sent=None*)

Raised when trying to interact with a closed connection.

rcvd

if a close frame was received, its code and reason are available in `rcvd.code` and `rcvd.reason`.

Type
Optional[*Close*]

sent

if a close frame was sent, its code and reason are available in `sent.code` and `sent.reason`.

Type
Optional[*Close*]

rcvd_then_sent

if close frames were received and sent, this attribute tells in which order this happened, from the perspective of this side of the connection.

Type
Optional[bool]

exception websockets.exceptions.**ConnectionClosedError**(*rcvd*, *sent*, *rcvd_then_sent=None*)

Like *ConnectionClosed*, when the connection terminated with an error.

A close code other than 1000 (OK) or 1001 (going away) was received or sent, or the closing handshake didn't complete properly.

exception websockets.exceptions.**ConnectionClosedOK**(*rcvd*, *sent*, *rcvd_then_sent=None*)

Like *ConnectionClosed*, when the connection terminated properly.

A close code 1000 (OK) or 1001 (going away) was received and sent.

exception websockets.exceptions.**DuplicateParameter**(*name*)

Raised when a parameter name is repeated in an extension header.

exception websockets.exceptions.**InvalidHandshake**

Raised during the handshake when the WebSocket connection fails.

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.**InvalidMessage**

Raised when a handshake request or response is malformed.

exception websockets.exceptions.**InvalidOrigin**(*origin*)

Raised when the Origin header in a request isn't allowed.

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.**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.**InvalidStatus**(*response*)

Raised when a handshake response rejects the WebSocket upgrade.

exception websockets.exceptions.**InvalidStatusCode**(*status_code*, *headers*)

Raised when a handshake response status code is invalid.

exception websockets.exceptions.**InvalidURI**(*uri*, *msg*)

Raised when connecting to an URI that isn't a valid WebSocket URI.

exception websockets.exceptions.**InvalidUpgrade**(*name*, *value=None*)

Raised when the Upgrade or Connection header isn't correct.

exception websockets.exceptions.**NegotiationError**

Raised when negotiating an extension fails.

exception websockets.exceptions.**PayloadTooBig**

Raised when receiving a frame with a payload exceeding the maximum size.

exception websockets.exceptions.**ProtocolError**

Raised when a frame breaks the protocol.

exception websockets.exceptions.**RedirectHandshake**(*uri*)

Raised when a handshake gets redirected.

This exception is an implementation detail.

exception websockets.exceptions.**SecurityError**

Raised when a handshake request or response breaks a security rule.

Security limits are hard coded.

exception websockets.exceptions.**WebSocketException**

Base class for all exceptions defined by websockets.

websockets.exceptions.**WebSocketProtocolError**

alias of *ProtocolError*

3.6 Types

websockets.typing.**Data**

Types supported in a WebSocket message: `str` for a Text frame, `bytes` for a Binary.

alias of `Union`[`str`, `bytes`]

websockets.typing.**LoggerLike**

Types accepted where a `Logger` is expected.

alias of `Union`[`Logger`, `LoggerAdapter`]

websockets.typing.**Origin**

Value of a `Origin` header.

alias of `str`

websockets.typing.**Subprotocol**

Subprotocol in a `Sec-WebSocket-Protocol` header.

alias of `str`

websockets.typing.**ExtensionName**

Name of a WebSocket extension.

alias of `str`

websockets.typing.**ExtensionParameter**

Parameter of a WebSocket extension.

alias of `Tuple[str, Optional[str]]`

websockets.connection.**Event**

Events that *events_received()* may return.

alias of `Union[Request, Response, Frame]`

websockets.datastructures.**HeadersLike**

Types accepted where *Headers* is expected.

alias of `Union[Headers, Mapping[str, str], Iterable[Tuple[str, str]]]`

3.7 Extensions

The WebSocket protocol supports extensions.

At the time of writing, there's only one registered extension with a public specification, WebSocket Per-Message Deflate.

3.7.1 Per-Message Deflate

websockets.extensions.permessage_deflate implements WebSocket Per-Message Deflate.

This extension is specified in **RFC 7692**.

Refer to the *topic guide on compression* to learn more about tuning compression settings.

class websockets.extensions.permessage_deflate.**ClientPerMessageDeflateFactory**(*server_no_context_takeover=Fal*
client_no_context_takeover=Fals
server_max_window_bits=None,
client_max_window_bits=True,
com-
press_settings=None)

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*) – prevent server from using context takeover.
- **client_no_context_takeover** (*bool*) – prevent client from using context takeover.

- **server_max_window_bits** (*Optional[int]*) – maximum size of the server's LZ77 sliding window in bits, between 8 and 15.
- **client_max_window_bits** (*Optional[Union[int, bool]]*) – maximum size of the client's LZ77 sliding window in bits, between 8 and 15, or `True` to indicate support without setting a limit.
- **compress_settings** (*Optional[Dict[str, Any]]*) – additional keyword arguments for `zlib.compressobj()`, excluding `wbits`.

class websockets.extensions.permessage_deflate.**ServerPerMessageDeflateFactory**(*server_no_context_takeover=Fal*
client_no_context_takeover=Fals
server_max_window_bits=None,
client_max_window_bits=None,
com-
press_settings=None)

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*) – prevent server from using context takeover.
- **client_no_context_takeover** (*bool*) – prevent client from using context takeover.
- **server_max_window_bits** (*Optional[int]*) – maximum size of the server's LZ77 sliding window in bits, between 8 and 15.
- **client_max_window_bits** (*Optional[int]*) – maximum size of the client's LZ77 sliding window in bits, between 8 and 15, or `True` to indicate support without setting a limit.
- **compress_settings** (*Optional[Dict[str, Any]]*) – additional keyword arguments for `zlib.compressobj()`, excluding `wbits`.

3.7.2 Base classes

websockets.extensions **defines base classes for implementing**
extensions.

Refer to the *how-to guide on extensions* to learn more about writing an extension.

class websockets.extensions.**Extension**

Base class for extensions.

name: *ExtensionName*

Extension identifier.

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

Decode an incoming frame.

Parameters

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

Returns
Decoded frame.

Return type
Frame

Raises
PayloadTooBig – if decoding the payload exceeds `max_size`.

encode(*frame*)

Encode an outgoing frame.

Parameters
frame (`Frame`) – outgoing frame.

Returns
Encoded frame.

Return type
Frame

class websockets.extensions.**ClientExtensionFactory**

Base class for client-side extension factories.

name: ***ExtensionName***

Extension identifier.

get_request_params()

Build parameters to send to the server for this extension.

Returns
Parameters to send to the server.

Return type
List[*ExtensionParameter*]

process_response_params(*params*, *accepted_extensions*)

Process parameters received from the server.

Parameters

- **params** (*Sequence[*`ExtensionParameter`*]*) – parameters received from the server for this extension.
- **accepted_extensions** (*Sequence[*`Extension`*]*) – list of previously accepted extensions.

Returns
An extension instance.

Return type
Extension

Raises
NegotiationError – if parameters aren't acceptable.

class websockets.extensions.**ServerExtensionFactory**

Base class for server-side extension factories.

process_request_params(*params*, *accepted_extensions*)

Process parameters received from the client.

Parameters

- **params** (*Sequence[ExtensionParameter]*) – parameters received from the client for this extension.
- **accepted_extensions** (*Sequence[Extension]*) – list of previously accepted extensions.

Returns
To accept the offer, parameters to send to the client for this extension and an extension instance.

Return type
Tuple[List[*ExtensionParameter*], *Extension*]

Raises
NegotiationError – to reject the offer, if parameters received from the client aren't acceptable.

3.8 Limitations

3.8.1 Client

The client doesn't attempt to guarantee that there is no more than one connection to a given IP address in a CONNECTING state. This behavior is mandated by RFC 6455. However, *connect()* isn't the right layer for enforcing this constraint. It's the caller's responsibility.

The client doesn't support connecting through a HTTP proxy (issue 364) or a SOCKS proxy (issue 475).

3.8.2 Server

At this time, there are no known limitations affecting only the server.

3.8.3 Both sides

There is no way to control compression of outgoing frames on a per-frame basis (issue 538). If compression is enabled, all frames are compressed.

There is no way to receive each fragment of a fragmented messages as it arrives (issue 479). websockets always reassembles fragmented messages before returning them.

Public API documented in the API reference are subject to the *backwards-compatibility policy*.

Anything that isn't listed in the API reference is a private API. There's no guarantees of behavior or backwards-compatibility for private APIs.

For convenience, many public APIs can be imported from the `websockets` package. This feature is incompatible with static code analysis tools such as mypy, though. If you're using such tools, use the full import path.

CHAPTER

FOUR

TOPIC GUIDES

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

4.1 Deployment

When you deploy your websockets server to production, at a high level, your architecture will almost certainly look like the following diagram:

The basic unit for scaling a websockets server is "one server process". Each blue box in the diagram represents one server process.

There's more variation in routing. While the routing layer is shown as one big box, it is likely to involve several subsystems.

When you design a deployment, your should consider two questions:

1. How will I run the appropriate number of server processes?
2. How will I route incoming connections to these processes?

These questions are strongly related. There's a wide range of acceptable answers, depending on your goals and your constraints.

You can find a few concrete examples in the *deployment how-to guides*.

4.1.1 Running server processes

How many processes do I need?

Typically, one server process will manage a few hundreds or thousands connections, depending on the frequency of messages and the amount of work they require.

CPU and memory usage increase with the number of connections to the server.

Often CPU is the limiting factor. If a server process goes to 100% CPU, then you reached the limit. How much headroom you want to keep is up to you.

Once you know how many connections a server process can manage and how many connections you need to handle, you can calculate how many processes to run.

You can also automate this calculation by configuring an autoscaler to keep CPU usage or connection count within acceptable limits.

Don't scale with threads. Threads doesn't make sense for a server built with `asyncio`.

How do I run processes?

Most solutions for running multiple instances of a server process fall into one of these three buckets:

1. Running N processes on a platform:
   - a Kubernetes Deployment
   - its equivalent on a Platform as a Service provider
2. Running N servers:
   - an AWS Auto Scaling group, a GCP Managed instance group, etc.
   - a fixed set of long-lived servers
3. Running N processes on a server:
   - preferably via a process manager or supervisor

Option 1 is easiest of you have access to such a platform.

Option 2 almost always combines with option 3.

How do I start a process?

Run a Python program that invokes *serve()*. That's it.

Don't run an ASGI server such as Uvicorn, Hypercorn, or Daphne. They're alternatives to websockets, not complements.

Don't run a WSGI server such as Gunicorn, Waitress, or mod_wsgi. They aren't designed to run WebSocket applications.

Applications servers handle network connections and expose a Python API. You don't need one because websockets handles network connections directly.

How do I stop a process?

Process managers send the SIGTERM signal to terminate processes. Catch this signal and exit the server to ensure a graceful shutdown.

Here's an example:

```
#!/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 server():
    # Set the stop condition when receiving SIGTERM.
    loop = asyncio.get_running_loop()
    stop = loop.create_future()
    loop.add_signal_handler(signal.SIGTERM, stop.set_result, None)
```

(continues on next page)

(continued from previous page)

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

asyncio.run(server())
```

When exiting the context manager, *serve()* closes all connections with code 1001 (going away). As a consequence:

- If the connection handler is awaiting *recv()*, it receives a *ConnectionClosedOK* exception. It can catch the exception and clean up before exiting.
- Otherwise, it should be waiting on *wait_closed()*, so it can receive the *ConnectionClosedOK* exception and exit.

This example is easily adapted to handle other signals.

If you override the default signal handler for SIGINT, which raises `KeyboardInterrupt`, be aware that you won't be able to interrupt a program with Ctrl-C anymore when it's stuck in a loop.

4.1.2 Routing connections

What does routing involve?

Since the routing layer is directly exposed to the Internet, it should provide appropriate protection against threats ranging from Internet background noise to targeted attacks.

You should always secure WebSocket connections with TLS. Since the routing layer carries the public domain name, it should terminate TLS connections.

Finally, it must route connections to the server processes, balancing new connections across them.

How do I route connections?

Here are typical solutions for load balancing, matched to ways of running processes:

1. If you're running on a platform, it comes with a routing layer:
   - a Kubernetes Ingress and Service
   - a service mesh: Istio, Consul, Linkerd, etc.
   - the routing mesh of a Platform as a Service
2. If you're running N servers, you may load balance with:
   - a cloud load balancer: AWS Elastic Load Balancing, GCP Cloud Load Balancing, etc.
   - A software load balancer: HAProxy, NGINX, etc.
3. If you're running N processes on a server, you may load balance with:
   - A software load balancer: HAProxy, NGINX, etc.
   - The operating system — all processes listen on the same port

You may trust the load balancer to handle encryption and to provide security. You may add another layer in front of the load balancer for these purposes.

There are many possibilities. Don't add layers that you don't need, though.

How do I implement a health check?

Load balancers need a way to check whether server processes are up and running to avoid routing connections to a non-functional backend.

websockets provide minimal support for responding to HTTP requests with the *process_request()* hook.

Here's an example:

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

import asyncio
import http
import websockets

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

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

async def main():
    async with websockets.serve(
        echo, "localhost", 8765,
        process_request=health_check,
    ):
        await asyncio.Future()  # run forever

asyncio.run(main())
```

4.2 Logging

4.2.1 Logs contents

When you run a WebSocket client, your code calls coroutines provided by websockets.

If an error occurs, websockets tells you by raising an exception. For example, it raises a *ConnectionClosed* exception if the other side closes the connection.

When you run a WebSocket server, websockets accepts connections, performs the opening handshake, runs the connection handler coroutine that you provided, and performs the closing handshake.

Given this inversion of control, if an error happens in the opening handshake or if the connection handler crashes, there is no way to raise an exception that you can handle.

Logs tell you about these errors.

Besides errors, you may want to record the activity of the server.

In a request/response protocol such as HTTP, there's an obvious way to record activity: log one event per request/response. Unfortunately, this solution doesn't work well for a bidirectional protocol such as WebSocket.

Instead, when running as a server, websockets logs one event when a connection is established and another event when a connection is closed.

By default, websockets doesn't log an event for every message. That would be excessive for many applications exchanging small messages at a fast rate. If you need this level of detail, you could add logging in your own code.

Finally, you can enable debug logs to get details about everything websockets is doing. This can be useful when developing clients as well as servers.

See *log levels* below for a list of events logged by websockets logs at each log level.

4.2.2 Configure logging

websockets relies on the `logging` module from the standard library in order to maximize compatibility and integrate nicely with other libraries:

```
import logging
```

websockets logs to the `"websockets.client"` and `"websockets.server"` loggers.

websockets doesn't provide a default logging configuration because requirements vary a lot depending on the environment.

Here's a basic configuration for a server in production:

```
logging.basicConfig(
    format="%(asctime)s %(message)s",
    level=logging.INFO,
)
```

Here's how to enable debug logs for development:

```
logging.basicConfig(
    format="%(message)s",
    level=logging.DEBUG,
)
```

Furthermore, websockets adds a `websocket` attribute to log records, so you can include additional information about the current connection in logs.

You could attempt to add information with a formatter:

```
# this doesn't work!
logging.basicConfig(
    format="{asctime} {websocket.id} {websocket.remote_address[0]} {message}",
    level=logging.INFO,
    style="{",
)
```

However, this technique runs into two problems:

- The formatter applies to all records. It will crash if it receives a record without a `websocket` attribute. For example, this happens when logging that the server starts because there is no current connection.
- Even with `str.format()` style, you're restricted to attribute and index lookups, which isn't enough to implement some fairly simple requirements.

There's a better way. *`connect()`* and *`serve()`* accept a `logger` argument to override the default `Logger`. You can set `logger` to a `LoggerAdapter` that enriches logs.

For example, if the server is behind a reverse proxy, *`remote_address`* gives the IP address of the proxy, which isn't useful. IP addresses of clients are provided in a HTTP header set by the proxy.

Here's how to include them in logs, assuming they're in the `X-Forwarded-For` header:

```
logging.basicConfig(
    format="%(asctime)s %(message)s",
    level=logging.INFO,
)

class LoggerAdapter(logging.LoggerAdapter):
    """Add connection ID and client IP address to websockets logs."""
    def process(self, msg, kwargs):
        try:
            websocket = kwargs["extra"]["websocket"]
        except KeyError:
            return msg, kwargs
        xff = websocket.request_headers.get("X-Forwarded-For")
        return f"{websocket.id} {xff} {msg}", kwargs

async with websockets.serve(
    ...,
    logger=LoggerAdapter(logging.getLogger("websockets.server")),
):
    ...
```

4.2.3 Logging to JSON

Even though `logging` predates structured logging, it's still possible to output logs as JSON with a bit of effort.

First, we need a `Formatter` that renders JSON:

```
import json
import logging
import datetime

class JSONFormatter(logging.Formatter):
    """
    Render logs as JSON.

    To add details to a log record, store them in a ``event_data``
    custom attribute. This dict is merged into the event.

    """
    def __init__(self):
        pass  # override logging.Formatter constructor

    def format(self, record):
        event = {
            "timestamp": self.getTimestamp(record.created),
            "message": record.getMessage(),
            "level": record.levelname,
            "logger": record.name,
        }
        event_data = getattr(record, "event_data", None)
        if event_data:
```

(continues on next page)

(continued from previous page)

```
            event.update(event_data)
        if record.exc_info:
            event["exc_info"] = self.formatException(record.exc_info)
        if record.stack_info:
            event["stack_info"] = self.formatStack(record.stack_info)
        return json.dumps(event)

    def getTimestamp(self, created):
        return datetime.datetime.utcfromtimestamp(created).isoformat()
```

Then, we configure logging to apply this formatter:

```
handler = logging.StreamHandler()
handler.setFormatter(formatter)

logger = logging.getLogger()
logger.addHandler(handler)
logger.setLevel(logging.INFO)
```

Finally, we populate the `event_data` custom attribute in log records with a `LoggerAdapter`:

```
class LoggerAdapter(logging.LoggerAdapter):
    """Add connection ID and client IP address to websockets logs."""
    def process(self, msg, kwargs):
        try:
            websocket = kwargs["extra"]["websocket"]
        except KeyError:
            return msg, kwargs
        kwargs["extra"]["event_data"] = {
            "connection_id": str(websocket.id),
            "remote_addr": websocket.request_headers.get("X-Forwarded-For"),
        }
        return msg, kwargs

async with websockets.serve(
    ...,
    logger=LoggerAdapter(logging.getLogger("websockets.server")),
):
    ...
```

4.2.4 Disable logging

If your application doesn't configure `logging`, Python outputs messages of severity `WARNING` and higher to `stderr`. As a consequence, you will see a message and a stack trace if a connection handler coroutine crashes or if you hit a bug in websockets.

If you want to disable this behavior for websockets, you can add a `NullHandler`:

```
logging.getLogger("websockets").addHandler(logging.NullHandler())
```

Additionally, if your application configures `logging`, you must disable propagation to the root logger, or else its handlers could output logs:

```
logging.getLogger("websockets").propagate = False
```

Alternatively, you could set the log level to `CRITICAL` for the `"websockets"` logger, as the highest level currently used is `ERROR`:

```
logging.getLogger("websockets").setLevel(logging.CRITICAL)
```

Or you could configure a filter to drop all messages:

```
logging.getLogger("websockets").addFilter(lambda record: None)
```

4.2.5 Log levels

Here's what websockets logs at each level.

ERROR

- Exceptions raised by connection handler coroutines in servers
- Exceptions resulting from bugs in websockets

INFO

- Server starting and stopping
- Server establishing and closing connections
- Client reconnecting automatically

DEBUG

- Changes to the state of connections
- Handshake requests and responses
- All frames sent and received
- Steps to close a connection
- Keepalive pings and pongs
- Errors handled transparently

Debug messages have cute prefixes that make logs easier to scan:

- `>` - send something
- `<` - receive something
- `=` - set connection state
- `x` - shut down connection
- `%` - manage pings and pongs
- `!` - handle errors and timeouts

4.3 Authentication

The WebSocket protocol was designed for creating web applications that need bidirectional communication between clients running in browsers and servers.

In most practical use cases, WebSocket servers need to authenticate clients in order to route communications appropriately and securely.

RFC 6455 stays elusive when it comes to authentication:

> This protocol doesn't prescribe any particular way that servers can authenticate clients during the WebSocket handshake. The WebSocket server can use any client authentication mechanism available to a generic HTTP server, such as cookies, HTTP authentication, or TLS authentication.

None of these three mechanisms works well in practice. Using cookies is cumbersome, HTTP authentication isn't supported by all mainstream browsers, and TLS authentication in a browser is an esoteric user experience.

Fortunately, there are better alternatives! Let's discuss them.

4.3.1 System design

Consider a setup where the WebSocket server is separate from the HTTP server.

Most servers built with websockets to complement a web application adopt this design because websockets doesn't aim at supporting HTTP.

The following diagram illustrates the authentication flow.

Assuming the current user is authenticated with the HTTP server (1), the application needs to obtain credentials from the HTTP server (2) in order to send them to the WebSocket server (3), who can check them against the database of user accounts (4).

Usernames and passwords aren't a good choice of credentials here, if only because passwords aren't available in clear text in the database.

Tokens linked to user accounts are a better choice. These tokens must be impossible to forge by an attacker. For additional security, they can be short-lived or even single-use.

4.3.2 Sending credentials

Assume the web application obtained authentication credentials, likely a token, from the HTTP server. There's four options for passing them to the WebSocket server.

1. **Sending credentials as the first message in the WebSocket connection.**

   This is fully reliable and the most secure mechanism in this discussion. It has two minor downsides:

   - Authentication is performed at the application layer. Ideally, it would be managed at the protocol layer.
   - Authentication is performed after the WebSocket handshake, making it impossible to monitor authentication failures with HTTP response codes.

2. **Adding credentials to the WebSocket URI in a query parameter.**

   This is also fully reliable but less secure. Indeed, it has a major downside:

   - URIs end up in logs, which leaks credentials. Even if that risk could be lowered with single-use tokens, it is usually considered unacceptable.

Authentication is still performed at the application layer but it can happen before the WebSocket handshake, which improves separation of concerns and enables responding to authentication failures with HTTP 401.

3. **Setting a cookie on the domain of the WebSocket URI.**

   Cookies are undoubtedly the most common and hardened mechanism for sending credentials from a web application to a server. In a HTTP application, credentials would be a session identifier or a serialized, signed session.

   Unfortunately, when the WebSocket server runs on a different domain from the web application, this idea bumps into the Same-Origin Policy. For security reasons, setting a cookie on a different origin is impossible.

   The proper workaround consists in:

   - creating a hidden iframe served from the domain of the WebSocket server
   - sending the token to the iframe with postMessage
   - setting the cookie in the iframe

   before opening the WebSocket connection.

   Sharing a parent domain (e.g. example.com) between the HTTP server (e.g. www.example.com) and the WebSocket server (e.g. ws.example.com) and setting the cookie on that parent domain would work too.

   However, the cookie would be shared with all subdomains of the parent domain. For a cookie containing credentials, this is unacceptable.

4. **Adding credentials to the WebSocket URI in user information.**

   Letting the browser perform HTTP Basic Auth is a nice idea in theory.

   In practice it doesn't work due to poor support in browsers.

   As of May 2021:

   - Chrome 90 behaves as expected.
   - Firefox 88 caches credentials too aggressively.

     When connecting again to the same server with new credentials, it reuses the old credentials, which may be expired, resulting in an HTTP 401. Then the next connection succeeds. Perhaps errors clear the cache.

     When tokens are short-lived or single-use, this bug produces an interesting effect: every other WebSocket connection fails.

   - Safari 14 ignores credentials entirely.

Two other options are off the table:

1. **Setting a custom HTTP header**

   This would be the most elegant mechanism, solving all issues with the options discussed above.

   Unfortunately, it doesn't work because the WebSocket API doesn't support setting custom headers.

2. **Authenticating with a TLS certificate**

   While this is suggested by the RFC, installing a TLS certificate is too far from the mainstream experience of browser users. This could make sense in high security contexts. I hope developers working on such projects don't take security advice from the documentation of random open source projects.

4.3.3 Let's experiment!

The experiments/authentication directory demonstrates these techniques.

Run the experiment in an environment where websockets is installed:

```
$ python experiments/authentication/app.py
Running on http://localhost:8000/
```

When you browse to the HTTP server at http://localhost:8000/ and you submit a username, the server creates a token and returns a testing web page.

This page opens WebSocket connections to four WebSocket servers running on four different origins. It attempts to authenticate with the token in four different ways.

First message

As soon as the connection is open, the client sends a message containing the token:

```
const websocket = new WebSocket("ws://.../");
websocket.onopen = () => websocket.send(token);

// ...
```

At the beginning of the connection handler, the server receives this message and authenticates the user. If authentication fails, the server closes the connection:

```
async def first_message_handler(websocket, path):
    token = await websocket.recv()
    user = get_user(token)
    if user is None:
        await websocket.close(1011, "authentication failed")
        return

    ...
```

Query parameter

The client adds the token to the WebSocket URI in a query parameter before opening the connection:

```
const uri = `ws://.../?token=${token}`;
const websocket = new WebSocket(uri);

// ...
```

The server intercepts the HTTP request, extracts the token and authenticates the user. If authentication fails, it returns a HTTP 401:

```
class QueryParamProtocol(websockets.WebSocketServerProtocol):
    async def process_request(self, path, headers):
        token = get_query_parameter(path, "token")
        if token is None:
            return http.HTTPStatus.UNAUTHORIZED, [], b"Missing token\n"
```

(continues on next page)

(continued from previous page)

```
        user = get_user(token)
        if user is None:
            return http.HTTPStatus.UNAUTHORIZED, [], b"Invalid token\n"

        self.user = user

async def query_param_handler(websocket, path):
    user = websocket.user

    ...
```

Cookie

The client sets a cookie containing the token before opening the connection.

The cookie must be set by an iframe loaded from the same origin as the WebSocket server. This requires passing the token to this iframe.

```
// in main window
iframe.contentWindow.postMessage(token, "http://...");

// in iframe
document.cookie = `token=${data}; SameSite=Strict`;

// in main window
const websocket = new WebSocket("ws://.../");

// ...
```

This sequence must be synchronized between the main window and the iframe. This involves several events. Look at the full implementation for details.

The server intercepts the HTTP request, extracts the token and authenticates the user. If authentication fails, it returns a HTTP 401:

```
class CookieProtocol(websockets.WebSocketServerProtocol):
    async def process_request(self, path, headers):
        # Serve iframe on non-WebSocket requests
        ...

        token = get_cookie(headers.get("Cookie", ""), "token")
        if token is None:
            return http.HTTPStatus.UNAUTHORIZED, [], b"Missing token\n"

        user = get_user(token)
        if user is None:
            return http.HTTPStatus.UNAUTHORIZED, [], b"Invalid token\n"

        self.user = user

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

(continues on next page)

(continued from previous page)

```
    user = websocket.user

    ...
```

User information

The client adds the token to the WebSocket URI in user information before opening the connection:

```
const uri = `ws://token:${token}@.../`;
const websocket = new WebSocket(uri);

// ...
```

Since HTTP Basic Auth is designed to accept a username and a password rather than a token, we send `token` as username and the token as password.

The server intercepts the HTTP request, extracts the token and authenticates the user. If authentication fails, it returns a HTTP 401:

```
class UserInfoProtocol(websockets.BasicAuthWebSocketServerProtocol):
    async def check_credentials(self, username, password):
        if username != "token":
            return False

        user = get_user(password)
        if user is None:
            return False

        self.user = user
        return True

async def user_info_handler(websocket, path):
    user = websocket.user

    ...
```

4.3.4 Machine-to-machine authentication

When the WebSocket client is a standalone program rather than a script running in a browser, there are far fewer constraints. HTTP Authentication is the best solution in this scenario.

To authenticate a websockets client with HTTP Basic Authentication (**RFC 7617**), include the credentials in the URI:

```
async with websockets.connect(
    f"wss://{username}:{password}@example.com",
) as websocket:
    ...
```

(You must `quote()` `username` and `password` if they contain unsafe characters.)

To authenticate a websockets client with HTTP Bearer Authentication (**RFC 6750**), add a suitable `Authorization` header:

```
async with websockets.connect(
    "wss://example.com",
    extra_headers={"Authorization": f"Bearer {token}"}
) as websocket:
    ...
```

4.4 Broadcasting messages

Note: If you just want to send a message to all connected clients, use *broadcast()*.

If you want to learn about its design in depth, continue reading this document.

WebSocket servers often send the same message to all connected clients or to a subset of clients for which the message is relevant.

Let's explore options for broadcasting a message, explain the design of *broadcast()*, and discuss alternatives.

For each option, we'll provide a connection handler called `handler()` and a function or coroutine called `broadcast()` that sends a message to all connected clients.

Integrating them is left as an exercise for the reader. You could start with:

```
import asyncio
import websockets

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

async def broadcast(message):
    ...

async def broadcast_messages():
    while True:
        await asyncio.sleep(1)
        message = ...  # your application logic goes here
        await broadcast(message)

async def main():
    async with websockets.serve(handler, "localhost", 8765):
        await broadcast_messages()  # runs forever

if __name__ == "__main__":
    asyncio.run(main())
```

`broadcast_messages()` must yield control to the event loop between each message, or else it will never let the server run. That's why it includes `await asyncio.sleep(1)`.

A complete example is available in the experiments/broadcast directory.

4.4.1 The naive way

The most obvious way to send a message to all connected clients consists in keeping track of them and sending the message to each of them.

Here's a connection handler that registers clients in a global variable:

```
CLIENTS = set()

async def handler(websocket, path):
    CLIENTS.add(websocket)
    try:
        await websocket.wait_closed()
    finally:
        CLIENTS.remove(websocket)
```

This implementation assumes that the client will never send any messages. If you'd rather not make this assumption, you can change:

```
await websocket.wait_closed()
```

to:

```
async for _ in websocket:
    pass
```

Here's a coroutine that broadcasts a message to all clients:

```
async def broadcast(message):
    for websocket in CLIENTS.copy():
        try:
            await websocket.send(message)
        except websockets.ConnectionClosed:
            pass
```

There are two tricks in this version of `broadcast()`.

First, it makes a copy of `CLIENTS` before iterating it. Else, if a client connects or disconnects while `broadcast()` is running, the loop would fail with:

```
RuntimeError: Set changed size during iteration
```

Second, it ignores *ConnectionClosed* exceptions because a client could disconnect between the moment `broadcast()` makes a copy of `CLIENTS` and the moment it sends a message to this client. This is fine: a client that disconnected doesn't belongs to "all connected clients" anymore.

The naive way can be very fast. Indeed, if all connections have enough free space in their write buffers, `await websocket.send(message)` writes the message and returns immediately, as it doesn't need to wait for the buffer to drain. In this case, `broadcast()` doesn't yield control to the event loop, which minimizes overhead.

The naive way can also fail badly. If the write buffer of a connection reaches `write_limit`, `broadcast()` waits for the buffer to drain before sending the message to other clients. This can cause a massive drop in performance.

As a consequence, this pattern works only when write buffers never fill up, which is usually outside of the control of the server.

If you know for sure that you will never write more than `write_limit` bytes within `ping_interval` + `ping_timeout`, then websockets will terminate slow connections before the write buffer has time to fill up.

Don't set extreme `write_limit`, `ping_interval`, and `ping_timeout` values to ensure that this condition holds. Set reasonable values and use the built-in *broadcast()* function instead.

4.4.2 The concurrent way

The naive way didn't work well because it serialized writes, while the whole point of asynchronous I/O is to perform I/O concurrently.

Let's modify `broadcast()` to send messages concurrently:

```
async def send(websocket, message):
    try:
        await websocket.send(message)
    except websockets.ConnectionClosed:
        pass

def broadcast(message):
    for websocket in CLIENTS:
        asyncio.create_task(send(websocket, message))
```

We move the error handling logic in a new coroutine and we schedule a `Task` to run it instead of executing it immediately.

Since `broadcast()` no longer awaits coroutines, we can make it a function rather than a coroutine and do away with the copy of `CLIENTS`.

This version of `broadcast()` makes clients independent from one another: a slow client won't block others. As a side effect, it makes messages independent from one another.

If you broadcast several messages, there is no strong guarantee that they will be sent in the expected order. Fortunately, the event loop runs tasks in the order in which they are created, so the order is correct in practice.

Technically, this is an implementation detail of the event loop. However, it seems unlikely for an event loop to run tasks in an order other than FIFO.

If you wanted to enforce the order without relying this implementation detail, you could be tempted to wait until all clients have received the message:

```
async def broadcast(message):
    if CLIENTS:  # asyncio.wait doesn't accept an empty list
        await asyncio.wait([
            asyncio.create_task(send(websocket, message))
            for websocket in CLIENTS
        ])
```

However, this doesn't really work in practice. Quite often, it will block until the slowest client times out.

4.4.3 Backpressure meets broadcast

At this point, it becomes apparent that backpressure, usually a good practice, doesn't work well when broadcasting a message to thousands of clients.

When you're sending messages to a single client, you don't want to send them faster than the network can transfer them and the client accept them. This is why *send()* checks if the write buffer is full and, if it is, waits until it drain, giving the network and the client time to catch up. This provides backpressure.

Without backpressure, you could pile up data in the write buffer until the server process runs out of memory and the operating system kills it.

The *send()* API is designed to enforce backpressure by default. This helps users of websockets write robust programs even if they never heard about backpressure.

For comparison, `asyncio.StreamWriter` requires users to understand backpressure and to await `drain()` explicitly after each `write()`.

When broadcasting messages, backpressure consists in slowing down all clients in an attempt to let the slowest client catch up. With thousands of clients, the slowest one is probably timing out and isn't going to receive the message anyway. So it doesn't make sense to synchronize with the slowest client.

How do we avoid running out of memory when slow clients can't keep up with the broadcast rate, then? The most straightforward option is to disconnect them.

If a client gets too far behind, eventually it reaches the limit defined by `ping_timeout` and websockets terminates the connection. You can read the discussion of *keepalive and timeouts* for details.

4.4.4 How `broadcast()` works

The built-in *broadcast()* function is similar to the naive way. The main difference is that it doesn't apply backpressure.

This provides the best performance by avoiding the overhead of scheduling and running one task per client.

Also, when sending text messages, encoding to UTF-8 happens only once rather than once per client, providing a small performance gain.

4.4.5 Per-client queues

At this point, we deal with slow clients rather brutally: we disconnect then.

Can we do better? For example, we could decide to skip or to batch messages, depending on how far behind a client is.

To implement this logic, we can create a queue of messages for each client and run a task that gets messages from the queue and sends them to the client:

```
import asyncio

CLIENTS = set()

async def relay(queue, websocket):
    while True:
        # Implement custom logic based on queue.qsize() and
        # websocket.transport.get_write_buffer_size() here.
        message = await queue.get()
        await websocket.send(message)
```

(continues on next page)

(continued from previous page)

```
async def handler(websocket, path):
    queue = asyncio.Queue()
    relay_task = asyncio.create_task(relay(queue, websocket))
    CLIENTS.add(queue)
    try:
        await websocket.wait_closed()
    finally:
        CLIENTS.remove(queue)
        relay_task.cancel()
```

Then we can broadcast a message by pushing it to all queues:

```
def broadcast(message):
    for queue in CLIENTS:
        queue.put_nowait(message)
```

The queues provide an additional buffer between the `broadcast()` function and clients. This makes it easier to support slow clients without excessive memory usage because queued messages aren't duplicated to write buffers until `relay()` processes them.

4.4.6 Publish–subscribe

Can we avoid centralizing the list of connected clients in a global variable?

If each client subscribes to a stream a messages, then broadcasting becomes as simple as publishing a message to the stream.

Here's a message stream that supports multiple consumers:

```
class PubSub:
    def __init__(self):
        self.waiter = asyncio.Future()

    def publish(self, value):
        waiter, self.waiter = self.waiter, asyncio.Future()
        waiter.set_result((value, self.waiter))

    async def subscribe(self):
        waiter = self.waiter
        while True:
            value, waiter = await waiter
            yield value

    __aiter__ = subscribe

PUBSUB = PubSub()
```

The stream is implemented as a linked list of futures. It isn't necessary to synchronize consumers. They can read the stream at their own pace, independently from one another. Once all consumers read a message, there are no references left, therefore the garbage collector deletes it.

The connection handler subscribes to the stream and sends messages:

```
async def handler(websocket, path):
    async for message in PUBSUB:
        await websocket.send(message)
```

The broadcast function publishes to the stream:

```
def broadcast(message):
    PUBSUB.publish(message)
```

Like per-client queues, this version supports slow clients with limited memory usage. Unlike per-client queues, it makes it difficult to tell how far behind a client is. The `PubSub` class could be extended or refactored to provide this information.

The `for` loop is gone from this version of the `broadcast()` function. However, there's still a `for` loop iterating on all clients hidden deep inside `asyncio`. When `publish()` sets the result of the `waiter` future, `asyncio` loops on callbacks registered with this future and schedules them. This is how connection handlers receive the next value from the asynchronous iterator returned by `subscribe()`.

4.4.7 Performance considerations

The built-in *`broadcast()`* function sends all messages without yielding control to the event loop. So does the naive way when the network and clients are fast and reliable.

For each client, a WebSocket frame is prepared and sent to the network. This is the minimum amount of work required to broadcast a message.

It would be tempting to prepare a frame and reuse it for all connections. However, this isn't possible in general for two reasons:

- Clients can negotiate different extensions. You would have to enforce the same extensions with the same parameters. For example, you would have to select some compression settings and reject clients that cannot support these settings.
- Extensions can be stateful, producing different encodings of the same message depending on previous messages. For example, you would have to disable context takeover to make compression stateless, resulting in poor compression rates.

All other patterns discussed above yield control to the event loop once per client because messages are sent by different tasks. This makes them slower than the built-in *`broadcast()`* function.

There is no major difference between the performance of per-message queues and publish–subscribe.

4.5 Compression

Most WebSocket servers exchange JSON messages because they're convenient to parse and serialize in a browser. These messages contain text data and tend to be repetitive.

This makes the stream of messages highly compressible. Enabling compression can reduce network traffic by more than 80%.

There's a standard for compressing messages. **RFC 7692** defines WebSocket Per-Message Deflate, a compression extension based on the Deflate algorithm.

4.5.1 Configuring compression

connect() and *serve()* enable compression by default because the reduction in network bandwidth is usually worth the additional memory and CPU cost.

If you want to disable compression, set `compression=None`:

```
import websockets

websockets.connect(..., compression=None)

websockets.serve(..., compression=None)
```

If you want to customize compression settings, you can enable the Per-Message Deflate extension explicitly with *ClientPerMessageDeflateFactory* or *ServerPerMessageDeflateFactory*:

```
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},
        ),
    ],
)

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

The Window Bits and Memory Level values in these examples reduce memory usage at the expense of compression rate.

4.5.2 Compression settings

When a client and a server enable the Per-Message Deflate extension, they negotiate two parameters to guarantee compatibility between compression and decompression. This affects the trade-off between compression rate and memory usage for both sides.

- **Context Takeover** means that the compression context is retained between messages. In other words, compression is applied to the stream of messages rather than to each message individually. Context takeover should remain enabled to get good performance on applications that send a stream of messages with the same structure, that is, most applications.

- **Window Bits** controls the size of the compression context. It must be an integer between 9 (lowest memory usage) and 15 (best compression). websockets defaults to 12. Setting it to 8 is possible but rejected by some versions of zlib.

`zlib` offers additional parameters for tuning compression. They control the trade-off between compression rate and CPU and memory usage for the compression side, transparently for the decompression side.

- **Memory Level** controls the size of the compression state. It must be an integer between 1 (lowest memory usage) and 9 (best compression). websockets defaults to 5. A lower memory level can increase speed thanks to memory locality.
- **Compression Level** controls the effort to optimize compression. It must be an integer between 1 (lowest CPU usage) and 9 (best compression).
- **Strategy** selects the compression strategy. The best choice depends on the type of data being compressed.

Unless mentioned otherwise, websockets uses the defaults of `zlib.compressobj()` for all these settings.

4.5.3 Tuning compression

By default, websockets enables compression with conservative settings that optimize memory usage at the cost of a slightly worse compression rate: Window Bits = 12 and Memory Level = 5. This strikes a good balance for small messages that are typical of WebSocket servers.

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.

Window Bits	Memory Level	Memory usage	Size vs. default	Time vs. default
15	8	322 KiB	-4.0%	+15%
14	7	178 KiB	-2.6%	+10%
13	6	106 KiB	-1.4%	+5%
12	**5**	**70 KiB**	**=**	**=**
11	4	52 KiB	+3.7%	-5%
10	3	43 KiB	+90%	+50%
9	2	39 KiB	+160%	+100%
—	—	19 KiB	+452%	—

Window Bits and Memory Level don't have to move in lockstep. However, other combinations don't yield significantly better results than those shown above.

Compressed size and compression time depend heavily on the kind of messages exchanged by the application so this example may not apply to your use case.

You can adapt compression/benchmark.py by creating a list of typical messages and passing it to the `_run` function.

Window Bits = 11 and Memory Level = 4 looks like the sweet spot in this table.

websockets defaults to Window Bits = 12 and Memory Level = 5 to stay away from Window Bits = 10 or Memory Level = 3 where performance craters, raising doubts on what could happen at Window Bits = 11 and Memory Level = 4 on a different corpus.

Defaults must be safe for all applications, hence a more conservative choice.

The benchmark focuses on compression because it's more expensive than decompression. Indeed, leaving aside small allocations, theoretical memory usage is:

- `(1 << (windowBits + 2)) + (1 << (memLevel + 9))` for compression;
- `1 << windowBits` for decompression.

CPU usage is also higher for compression than decompression.

4.5.4 Further reading

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 recommends Window Bits = 11 and Memory Level = 4 for optimizing memory usage.

4.6 Timeouts

Since the WebSocket protocol is intended for real-time communications over long-lived connections, it is desirable to ensure that connections don't break, and if they do, to report the problem quickly.

WebSocket is built on top of HTTP/1.1 where connections are short-lived, even with `Connection: keep-alive`. Typically, HTTP/1.1 infrastructure closes idle connections after 30 to 120 seconds.

As a consequence, proxies may terminate WebSocket connections prematurely, when no message was exchanged in 30 seconds.

In order to avoid this problem, websockets implements a keepalive mechanism based on WebSocket Ping and Pong frames. Ping and Pong are designed for this purpose.

By default, websockets waits 20 seconds, then sends a Ping frame, and expects to receive the corresponding Pong frame within 20 seconds. Else, it considers the connection broken and closes it.

Timings are configurable with the `ping_interval` and `ping_timeout` arguments of *`connect()`* and *`serve()`*.

While WebSocket runs on top of TCP, websockets doesn't rely on TCP keepalive because it's disabled by default and, if enabled, the default interval is no less than two hours, which doesn't meet requirements.

Latency between a client and a server may increase for two reasons:

- Network connectivity is poor. When network packets are lost, TCP attempts to retransmit them, which manifests as latency. Excessive packet loss makes the connection unusable in practice. At some point, timing out is a reasonable choice.
- Traffic is high. For example, if a client sends messages on the connection faster than a server can process them, this manifests as latency as well, because data is waiting in flight, mostly in OS buffers.

  If the server is more than 20 seconds behind, it doesn't see the Pong before the default timeout elapses. As a consequence, it closes the connection. This is a reasonable choice to prevent overload.

  If traffic spikes cause unwanted timeouts and you're confident that the server will catch up eventually, you can increase `ping_timeout` or you can disable keepalive entirely with `ping_interval=None`.

  The same reasoning applies to situations where the server sends more traffic than the client can accept.

You can monitor latency as follows:

```
import asyncio
import logging
import time

async def log_latency(websocket, logger):
    t0 = time.perf_counter()
    pong_waiter = await websocket.ping()
```

(continues on next page)

(continued from previous page)

```
    await pong_waiter
    t1 = time.perf_counter()
    logger.info("Connection latency: %.3f seconds", t1 - t0)

asyncio.create_task(log_latency(websocket, logging.getLogger()))
```

4.7 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*.

4.7.1 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.

4.7.2 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*.

4.7.3 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.

4.7.4 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.

4.7.5 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.

4.7.6 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.

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

4.7.8 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.

4.7.9 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.

4.7.10 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.

4.8 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.

4.8.1 Baseline

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

Refer to the *topic guide on compression* to learn more about tuning compression settings.

4.8.2 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 usage.

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*.

4.9 Security

4.9.1 Encryption

For production use, a server should require encrypted connections.

See this example of *encrypting connections with TLS*.

4.9.2 Memory usage

> **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 70 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 usage.

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

4.9.3 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 these limits, you can monkey-patch the constants in `websockets.http11`.

4.10 Performance

Here are tips to optimize performance.

4.10.1 uvloop

You can make a websockets application faster by running it with uvloop.

(This advice isn't specific to websockets. It applies to any `asyncio` application.)

4.10.2 broadcast

`broadcast()` is the most efficient way to send a message to many clients.

CHAPTER

FIVE

ABOUT WEBSOCKETS

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

5.1 Changelog

5.1.1 Backwards-compatibility policy

websockets is intended for production use. Therefore, stability is a goal.

websockets also aims at providing the best API for WebSocket in Python.

While we value stability, we value progress more. When an improvement requires changing a public API, we make the change and document it in this changelog.

When possible with reasonable effort, we preserve backwards-compatibility for five years after the release that introduced the change.

When a release contains backwards-incompatible API changes, the major version is increased, else the minor version is increased. Patch versions are only for fixing regressions shortly after a release.

Only documented APIs are public. Undocumented APIs are considered private. They may change at any time.

5.1.2 10.0

September 9, 2021

Backwards-incompatible changes

websockets 10.0 requires Python 3.7.

websockets 9.1 is the last version supporting Python 3.6.

The `loop` parameter is deprecated from all APIs.

This reflects a decision made in Python 3.8. See the release notes of Python 3.10 for details.

The `loop` parameter is also removed from *`WebSocketServer`*. This should be transparent.

`connect()` times out after 10 seconds by default.

You can adjust the timeout with the `open_timeout` parameter. Set it to `None` to disable the timeout entirely.

The `legacy_recv` option is deprecated.

See the release notes of websockets 3.0 for details.

The signature of `ConnectionClosed` changed.

If you raise *`ConnectionClosed`* or a subclass, rather than catch them when websockets raises them, you must change your code.

A `msg` parameter was added to `InvalidURI`.

If you raise *`InvalidURI`*, rather than catch it when websockets raises it, you must change your code.

New features

websockets 10.0 introduces a Sans-I/O API for easier integration in third-party libraries.

If you're integrating websockets in a library, rather than just using it, look at the *Sans-I/O integration guide*.

- Added compatibility with Python 3.10.
- Added *`broadcast()`* to send a message to many clients.
- Added support for reconnecting automatically by using *`connect()`* as an asynchronous iterator.
- Added `open_timeout` to *`connect()`*.
- Documented how to integrate with Django.
- Documented how to deploy websockets in production, with several options.
- Documented how to authenticate connections.
- Documented how to broadcast messages to many connections.

Improvements

- Improved logging. See the *logging guide*.
- Optimized default compression settings to reduce memory usage.
- Optimized processing of client-to-server messages when the C extension isn't available.
- Supported relative redirects in *`connect()`*.
- Handled TCP connection drops during the opening handshake.
- Made it easier to customize authentication with *`check_credentials()`*.
- Provided additional information in *`ConnectionClosed`* exceptions.
- Clarified several exceptions or log messages.

- Restructured documentation.
- Improved API documentation.
- Extended FAQ.

Bug fixes

- Avoided a crash when receiving a ping while the connection is closing.

5.1.3 9.1

May 27, 2021

Security fix

websockets 9.1 fixes a security issue introduced in 8.0.

Version 8.0 was vulnerable to timing attacks on HTTP Basic Auth passwords (CVE-2021-33880).

5.1.4 9.0.2

May 15, 2021

Bug fixes

- Restored compatibility of `python -m websockets` with Python < 3.9.
- Restored compatibility with mypy.

5.1.5 9.0.1

May 2, 2021

Bug fixes

- Fixed issues with the packaging of the 9.0 release.

5.1.6 9.0

May 1, 2021

Backwards-incompatible changes

Several modules are moved or deprecated.

Aliases provide compatibility for all previously public APIs according to the *backwards-compatibility policy*

- *Headers* and *MultipleValuesError* are moved from `websockets.http` to *websockets.datastructures*. If you're using them, you should adjust the import path.
- The `client`, `server`, `protocol`, and `auth` modules were moved from the `websockets` package to a `websockets.legacy` sub-package. Despite the name, they're still fully supported.
- The `framing`, `handshake`, `headers`, `http`, and `uri` modules in the `websockets` package are deprecated. These modules provided low-level APIs for reuse by other projects, but they didn't reach that goal. Keeping these APIs public makes it more difficult to improve websockets.

These changes pave the path for a refactoring that should be a transparent upgrade for most uses and facilitate integration by other projects.

Convenience imports from `websockets` are performed lazily.

While Python supports this, static code analysis tools such as mypy are unable to understand the behavior.

If you depend on such tools, use the real import path, which can be found in the API documentation, for example:

```
from websockets.client import connect
from websockets.server import serve
```

New features

- Added compatibility with Python 3.9.

Improvements

- Added support for IRIs in addition to URIs.
- Added close codes 1012, 1013, and 1014.
- Raised an error when passing a `dict` to *send()*.
- Improved error reporting.

Bug fixes

- Fixed sending fragmented, compressed messages.
- Fixed `Host` header sent when connecting to an IPv6 address.
- Fixed creating a client or a server with an existing Unix socket.
- Aligned maximum cookie size with popular web browsers.
- Ensured cancellation always propagates, even on Python versions where `CancelledError` inherits `Exception`.

5.1.7 8.1

November 1, 2019

New features

- Added compatibility with Python 3.8.

5.1.8 8.0.2

July 31, 2019

Bug fixes

- Restored the ability to pass a socket with the `sock` parameter of *`serve()`*.
- Removed an incorrect assertion when a connection drops.

5.1.9 8.0.1

July 21, 2019

Bug fixes

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

5.1.10 8.0

July 7, 2019

Backwards-incompatible changes

websockets 8.0 requires Python 3.6.

websockets 7.0 is the last version supporting Python 3.4 and 3.5.

`process_request` is now expected to be 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`. Previously, both were supported.

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

`max_queue` must be `None` to disable the limit.

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

The `host`, `port`, and `secure` attributes of `WebSocketCommonProtocol` are deprecated.

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

`WebSocketProtocolError` is renamed to `ProtocolError`.

An alias provides backwards compatibility.

`read_response()` now returns the reason phrase.

If you're using this low-level API, you must change your code.

New features

- 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.
- Added support for asynchronous generators in *`send()`* to generate fragmented messages incrementally.
- 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.

Improvements

- *`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.
- Changed *`WebSocketServer.close()`* to perform a proper closing handshake instead of failing the connection.
- Improved error messages when HTTP parsing fails.
- Improved API documentation.

Bug fixes

- 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.
- Avoided a crash when a `extra_headers` callable returns `None`.

5.1.11 7.0

November 1, 2018

Backwards-incompatible changes

Keepalive is enabled by default.

websockets now sends Ping frames at regular intervals and closes the connection if it doesn't receive a matching Pong frame. See *WebSocketCommonProtocol* for details.

Termination of connections by `WebSocketServer.close()` changes.

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())
```

Calling `recv()` concurrently raises a `RuntimeError`.

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

The `timeout` argument of `serve()` and `connect()` is renamed to `close_timeout`.

This prevents confusion with `ping_timeout`.

For backwards compatibility, `timeout` is still supported.

The `origins` argument of `serve()` changes.

Include `None` in the list rather than `''` to allow requests that don't contain an Origin header.

Pending pings aren't canceled when the connection is closed.

A ping — as in `ping = await websocket.ping()` — for which no pong was received yet used to be canceled when the connection is closed, so that `await ping` raised `CancelledError`.

Now `await ping` raises *ConnectionClosed* like other public APIs.

New features

- Added `process_request` and `select_subprotocol` arguments to *serve()* and *WebSocketServerProtocol* to facilitate customization of *process_request()* and *select_subprotocol()*.
- Added support for sending fragmented messages.
- Added the *wait_closed()* method to protocols.
- Added an interactive client: `python -m websockets <uri>`.

Improvements

- Improved handling of multiple HTTP headers with the same name.
- Improved error messages when a required HTTP header is missing.

Bug fixes

- Fixed a data loss bug in *recv()*: canceling it at the wrong time could result in messages being dropped.

5.1.12 6.0

July 16, 2018

Backwards-incompatible changes

The `Headers` class is introduced and several APIs are updated to use it.

- The `request_headers` argument of *process_request()* is now a *Headers* instead of an `http.client.HTTPMessage`.
- The `request_headers` and `response_headers` attributes of *WebSocketCommonProtocol* are now *Headers* instead of `http.client.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 `http.client.HTTPMessage` provide similar APIs, much of the code dealing with HTTP headers won't require changes.

New features

- Added compatibility with Python 3.7.

5.1.13 5.0.1

May 24, 2018

Bug fixes

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

5.1.14 5.0

May 22, 2018

Security fix

websockets 5.0 fixes a security issue introduced in 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).

Backwards-incompatible changes

A `user_info` field is added 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.

New features

- *connect()* performs HTTP Basic Auth when the URI contains credentials.
- *unix_serve()* can be used as an asynchronous context manager on Python 3.5.1.
- Added the *closed* property to protocols.
- Added new examples in the documentation.

Improvements

- 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.
- If a *ping()* doesn't receive a pong, it's canceled when the connection is closed.
- Reported the cause of *ConnectionClosed* exceptions.
- 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.
- Updated documentation with new features from Python 3.6.
- Improved several sections of the documentation.

Bug fixes

- Prevented `TypeError` due to missing close code on connection close.
- Fixed a race condition in the closing handshake that raised *InvalidState*.

5.1.15 4.0.1

November 2, 2017

Bug fixes

- Fixed issues with the packaging of the 4.0 release.

5.1.16 4.0

November 2, 2017

Backwards-incompatible changes

websockets 4.0 requires Python 3.4.

websockets 3.4 is the last version supporting Python 3.3.

Compression is enabled by default.

In August 2017, Firefox and Chrome support the permessage-deflate extension, 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()*.

The `state_name` attribute of protocols is deprecated.

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

New features

- *`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()`*.
- Allowed `extra_headers` to override `Server` and `User-Agent` headers.

Improvements

- Reorganized and extended documentation.
- Rewrote connection termination to increase robustness in edge cases.
- Reduced verbosity of "Failing the WebSocket connection" logs.

Bug fixes

- Aborted connections if they don't close within the configured `timeout`.
- Stopped leaking pending tasks when `cancel()` is called on a connection while it's being closed.

5.1.17 3.4

August 20, 2017

Backwards-incompatible changes

`InvalidStatus` is replaced by `InvalidStatusCode`.

This exception is raised when *`connect()`* receives an invalid response status code from the server.

New features

- *`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.

Improvements

- 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.
- Rewrote HTTP handling for simplicity and performance.
- Added an optional C extension to speed up low-level operations.

Bug fixes

- Providing a `sock` argument to *connect()* no longer crashes.

5.1.18 3.3

March 29, 2017

New features

- Ensured compatibility with Python 3.6.

Improvements

- Reduced noise in logs caused by connection resets.

Bug fixes

- Avoided crashing on concurrent writes on slow connections.

5.1.19 3.2

August 17, 2016

New features

- Added `timeout`, `max_size`, and `max_queue` arguments to *connect()* and *serve()*.

Improvements

- Made server shutdown more robust.

5.1.20 3.1

April 21, 2016

New features

- Added flow control for incoming data.

Bug fixes

- Avoided a warning when closing a connection before the opening handshake.

5.1.21 3.0

December 25, 2015

Backwards-incompatible changes

recv() now raises an exception when the connection is closed.

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, 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*.

New features

- *connect()* can be used as an asynchronous context manager on Python 3.5.1.
- *ping()* and *pong()* support data passed as str in addition to bytes.
- Made `state_name` attribute on protocols a public API.

Improvements

- Updated documentation with `await` and `async` syntax from Python 3.5.
- Worked around an `asyncio` bug affecting connection termination under load.
- Improved documentation.

5.1.22 2.7

November 18, 2015

New features

- Added compatibility with Python 3.5.

Improvements

- Refreshed documentation.

5.1.23 2.6

August 18, 2015

New features

- Added `local_address` and `remote_address` attributes on protocols.
- Closed open connections with code 1001 when a server shuts down.

Bug fixes

- Avoided TCP fragmentation of small frames.

5.1.24 2.5

July 28, 2015

New features

- Provided access to handshake request and response HTTP headers.
- Allowed customizing handshake request and response HTTP headers.
- Added support for running on a non-default event loop.

Improvements

- Improved documentation.
- Sent a 403 status code instead of 400 when request Origin isn't allowed.
- Clarified that the closing handshake can be initiated by the client.
- Set the close code and reason more consistently.
- Strengthened connection termination.

Bug fixes

- Canceling *recv()* no longer drops the next message.

5.1.25 2.4

January 31, 2015

New features

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

5.1.26 2.3

November 3, 2014

Improvements

- Improved compliance of close codes.

5.1.27 2.2

July 28, 2014

New features

- Added support for limiting message size.

5.1.28 2.1

April 26, 2014

New features

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

5.1.29 2.0

February 16, 2014

Backwards-incompatible changes

`send()`, `ping()`, and `pong()` are now coroutines.

They used to be functions.

Instead of:

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

you must write:

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

New features

- Added flow control for outgoing data.

5.1.30 1.0

November 14, 2013

New features

- Initial public release.

5.2 Contributing

Thanks for taking the time to contribute to websockets!

5.2.1 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)

5.2.2 Contributions

Bug reports, patches and suggestions are welcome!

Please open an issue or send a pull request.

Feedback about the documentation is especially valuable, as the primary author feels 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.

5.2.3 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 in the documentation;
- it wasn't asked already.

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

5.2.4 Cryptocurrency users

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

I'm aware of efforts to build proof-of-stake models. I'll care once the total carbon footprint of all cryptocurrencies drops to a non-bullshit level.

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 you.

I will summarily close issues related to Bitcoin or cryptocurrency in any way. Thanks for your understanding.

5.3 License

5.4 websockets for enterprise

5.4.1 Available as part of the Tidelift Subscription

Tidelift is working with the maintainers of websockets and thousands of other open source projects to deliver commercial support and maintenance for the open source dependencies you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact dependencies you use.

5.4.2 Enterprise-ready open source software—managed for you

The Tidelift Subscription is a managed open source subscription for application dependencies covering millions of open source projects across JavaScript, Python, Java, PHP, Ruby, .NET, and more.

Your subscription includes:

- **Security updates**
  - Tidelift's security response team coordinates patches for new breaking security vulnerabilities and alerts immediately through a private channel, so your software supply chain is always secure.
- **Licensing verification and indemnification**
  - Tidelift verifies license information to enable easy policy enforcement and adds intellectual property indemnification to cover creators and users in case something goes wrong. You always have a 100% up-to-date bill of materials for your dependencies to share with your legal team, customers, or partners.
- **Maintenance and code improvement**
  - Tidelift ensures the software you rely on keeps working as long as you need it to work. Your managed dependencies are actively maintained and we recruit additional maintainers where required.
- **Package selection and version guidance**
  - We help you choose the best open source packages from the start—and then guide you through updates to stay on the best releases as new issues arise.
- **Roadmap input**
  - Take a seat at the table with the creators behind the software you use. Tidelift's participating maintainers earn more income as their software is used by more subscribers, so they're interested in knowing what you need.
- **Tooling and cloud integration**
  - Tidelift works with GitHub, GitLab, BitBucket, and more. We support every cloud platform (and other deployment targets, too).

The end result? All of the capabilities you expect from commercial-grade software, for the full breadth of open source you use. That means less time grappling with esoteric open source trivia, and more time building your own applications—and your business.

PYTHON MODULE INDEX

W

INDEX

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W