aiocometd Documentation

Release 0.4.4

Robert Marki

Feb 26, 2019

Contents

aiocometd is a CometD client built using asyncio, implementing the Bayeux protocol.

CometD is a scalable WebSocket and HTTP based event and message routing bus. CometD makes use of WebSocket and HTTP push technologies known as Comet to provide low-latency data from the server to browsers and client applications.

CHAPTER 1

Features

- **Supported transports:**
  - `long-polling`
  - `websocket`
- Automatic reconnection after network failures
- Extensions

CHAPTER 2

Usage

```
import asyncio

from aiocometd import Client

async def chat():
    nickname = "John"

    # connect to the server
    async with Client("http://example.com/cometd") as client:

            # subscribe to channels to receive chat messages and
            # notifications about new members
            await client.subscribe("/chat/demo")
            await client.subscribe("/members/demo")

            # send initial message
            await client.publish("/chat/demo", {
                "user": nickname,
                "membership": "join",
                "chat": nickname + " has joined"
            })
            # add the user to the chat room's members
            await client.publish("/service/members", {
                "user": nickname,
                "room": "/chat/demo"
            })

            # listen for incoming messages
            async for message in client:
                if message["channel"] == "/chat/demo":
                    data = message["data"]
                    print(f"{data['user']}: {data['chat']}")

if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    loop = asyncio.get_event_loop()
    loop.run_until_complete(chat())
```

For more detailed usage examples take a look at the command line chat example or for a more complex example with a GUI check out the aiocometd-chat-demo.

CHAPTER 3

Documentation

https://aiocometd.readthedocs.io/

CHAPTER 4

Contents

4.1 User's guide

4.1.1 Installation

```
pip install aiocometd
```

Install extras

aiocometd defines several groups of optional requirements:

- `tests` for running unit tests
- `docs` for buidling the documentation
- `examples` for running the examples
- `dev` for creating a complete development enviroment

Any combination of these options can be specified during installation.

```
pip install aiocometd[tests,docs,examples,dev]
```

4.1.2 Quickstart

`Client` is the main interface of the library. It can be used to to connect to CometD servers, and to send and receive messages.

Connecting

After creating a *Client* object the *open()* method should be called to establish a connection with the server. The connection is closed and the session is terminated by calling the *close()* method.

```
client = Client("http://example.com/cometd")
await client.open()
# send and receive messsages...
await client.close()
```

Client objects can be also used as asynchronous context managers.

```
async with Client("http://example.com/cometd") as client:
    # send and receive messsages...
```

Channels

A channel is a string that looks like a URL path such as `/foo/bar`, `/meta/connect` or `/service/chat`.

The Bayeux specification defines three types of channels: *meta channels*, *service channels* and *broadcast channels*.

A channel that starts with `/meta/` is a meta channel, a channel that starts with `/service/` is a service channel, and all other channels are broadcast channels.

Meta channels

Meta channels provide to applications information about the Bayeux protocol, they are handled by the client internally, the users of the client shouldn't send or receive messages from these channels.

Service channels

Applications create *service channels*, which are used in the case of request/response style of communication between client and server (as opposed to the publish/subscribe style of communication of *broadcast channels*, see below). A server directly responds to messages sent to these channels, the sent message is not broadcasted to any other client.

Broadcast channels

Applications also create *broadcast channels*, which have the semantic of a messaging topic and are used in the case of the publish/subscribe style of communication, where one sender wants to broadcast information to multiple clients.

Subscriptions

In order to receive messages from *broadcast channels* a client must subscribe to these channels first.

```
await client.subscribe("/chat/demo")
```

If you no longer want to receive messages from one of the channels you're subscribed to then you must unsubscribe from the channel.

```
await client.unsubscribe("/chat/demo")
```

The current set of subscriptions can be obtained from the *Client.subscriptions* attribute.

Receiving messages

To receive messages broadcasted by the server after *subscribing* to these *channels* the *receive()* method should be used.

```
message = await client.receive()
```

The *receive()* method will wait until a message is received or it will raise a *TransportTimeoutError* in case the connection is lost with the server and the client can't re-establish the connection or a *ServerError* if the connection gets closed by the server.

The client can also be used as an asynchronous iterator in a for loop to wait for incoming messages.

```
async for message in client:
    # process message
```

Sending messages

To send messages to *service* or *broadcast* channels the *publish()* method can be used.

```
data = {"foo": "bar"}
response = await client.publish("/foo/bar", data)
```

4.1.3 Advanced Usage

Connection types

The Bayeux protocol used by CometD is a transport-independent protocol, that can be carried over HTTP or over WebSocket (or other transport protocols), so that an application is not bound to a specific transport technology.

aiocometd supports the *LONG_POLLING* and *WEBSOCKET* transports.

When a client connects to a CometD server, a so called handshake operation is executed first using the default transport that all CometD servers should support. Based on the types of transports that the server offers and what the client supports, the client picks one of the transports that it will use to communicate with the server.

By default, if the preferred connection types are not specified when the *Client* is created, it will use the *WEBSOCKET* transport if it's supported by the server or otherwise fall back to using *LONG_POLLING*.

If you prefer a different ordering then it can be specified when the *Client* is created:

```
client = Client("http://example.com/cometd",
                connection_types=[ConnectionType.LONG_POLLING,
                                  ConnectionType.WEBSOCKET])
```

If there is only a single connection type that you would wan't your client to accept or fail if it's not available on the server, then instead of a list specify a single connection type:

```
client = Client("http://example.com/cometd",
                connection_types=ConnectionType.WEBSOCKET)
```

Extensions

Extensions allow the modification of a message just after receiving it but before the rest of the message processing takes place, or just before sending it. An extension normally adds fields to the message being sent or received in the

ext object that the Bayeux protocol specification defines. An extension is not a way to add business fields to a message, but rather a way to process all messages, including the meta messages the Bayeux protocol uses, and to extend the Bayeux protocol itself.

aiocometd provides abstract base classes for implementing custom extensions using the *Extension* and *AuthExtension* classes.

Extension

To create a new extension use the *Extension* class as the base class:

```
class MyExtension(Extension):
    async def incoming(payload, headers=None):
        pass

    async def outgoing(payload, headers):
        pass
```

The incoming message payload, which is a list of messages, is first passed to the *incoming()* method along with the received headers. The incoming headers might or might not be empty, it depends on the type of transport used, whether it receives headers for responses.

The outgoing payload along with the headers are passed to the *outgoing()* method before sending.

Custom extension implementation can use these two methods to inspect or alter the messages or headers. The list of extension objects that you would want to use should be passed to the *Client*.

```
client = Client("http://example.com/cometd",
                extensions=[MyExtension()])
```

AuthExtension

The *AuthExtension* class, which is based on *Extension*, can be used to implement authentication extensions.

For authentication schemes where the credentials are static it doesn't makes much sense to use *AuthExtension* instead of *Extension*. However for schemes where the credentials can expire (like OAuth, JWT...) *authenticate()* method can be reimplemented to update those credentials. The *authenticate()* method is called by the client after an authentication failure.

```
class MyAuthExtension(AuthExtension):
    async def incoming(payload, headers=None):
        pass

    async def outgoing(payload, headers):
        pass

    async def authenticate():
        # get new JWT
```

An auth extension should be passed to the client separately from the other extensions.

```
client = Client("http://example.com/cometd",
                extensions=[MyExtension()]
                auth=MyAuthExtension())
```

Network failures

When a *Client* object is opened, it will try to maintain a continuous connection in the background with the server. If any network failures happen while waiting to *receive()* messages, the client will reconnect to the server transparently, it will resubscribe to the subscribed channels, and continue to wait for incoming messages.

To avoid waiting for a server which went offline permanently, a `connection_timeout` can be passed to the *Client*, to limit how many seconds the client object should wait before raising a *TransportTimeoutError* if it can't reconnect to the server.

```
client = Client("http://example.com/cometd",
                connection_timeout=60)

try:
    message = await client.receive()
except TransportTimeoutError:
    print("Connection is lost with the server. "
          "Couldn't reconnect in 60 seconds.")
```

The defaul value is `10` seconds. If you pass `None` as the `connection_timeout` value, then the client will keep on trying indefinitely.

Prefetch and backpressure

When a *Client* is opened it will start and maintain a connection in the background with the server. It will start to fetch messages from the server as soon as it's connected, even before *receive()* is called.

Firstly, prefetching messages has the advantage, that incoming messages will wait in a buffer for users to consume them when *receive()* is called, without any delay. Secondly, the client has no choice but to accept incoming messages.

The Bayeux protocol is modelled very heavily around long-polling type HTTP transports. Which requires from clients to send periodic requests to the server to simulate a continuous connection, otherwise the server will terminate the session. This makes it impossible to use backpressure, even with the type of transports like WebSocket which would otherwise support it. So the connection can not be suspended if the client can't keep up with receiving the incoming messages, or otherwise the session will be closed.

To avoid consuming all the available memory by the incoming messages, which are not consumed yet, the number of prefetched messages can be limited with the `max_pending_count` parameter of the *Client*. The default value is `100`.

```
client = Client("http://example.com/cometd",
                max_pending_count=42)
```

The current number of messages waiting to be consumed can be obtained from the *Client.pending_count* attribute.

JSON encoder/decoder

Besides the standard `json` module, many third party libraries offer JSON serialization/deserilization functionality. To use a different library for handling JSON data types, you can specify the callable to use for serialization with the `json_dumps` and the callable for deserialization with the `json_loads` parameters of the *Client*.

```
import ujson

```

(continues on next page)

(continued from previous page)

```
client = Client("http://example.com/cometd",
                json_dumps=ujson.dumps,
                json_loads=ujson.loads)
```

4.2 API Reference

4.2.1 Client

class aiocometd.**Client**(*url*, *connection_types=None*, ***, *connection_timeout=10.0*, *ssl=None*, *max_pending_count=100*, *extensions=None*, *auth=None*, *json_dumps=<function dumps>*, *json_loads=<function loads>*, *loop=None*)

CometD client

Parameters

- **url** (str) – CometD service url
- **connection_types** (Union[ConnectionType, List[ConnectionType], None]) – List of connection types in order of preference, or a single connection type name. If None, [*WEBSOCKET*, *LONG_POLLING*] will be used as a default value.
- **connection_timeout** (Union[int, float]) – The maximum amount of time to wait for the transport to re-establish a connection with the server when the connection fails.
- **ssl** (Union[SSLContext, bool, Fingerprint, None]) – SSL validation mode. None for default SSL check (ssl.create_default_context() is used), False for skip SSL certificate validation, aiohttp.Fingerprint for fingerprint validation, ssl.SSLContext for custom SSL certificate validation.
- **max_pending_count** (int) – The maximum number of messages to prefetch from the server. If the number of prefetched messages reach this size then the connection will be suspended, until messages are consumed. If it is less than or equal to zero, the count is infinite.
- **extensions** (Optional[List[Extension]]) – List of protocol extension objects
- **auth** (Optional[AuthExtension]) – An auth extension
- **json_dumps** (Callable[[Dict[str, Any]], str]) – Function for JSON serialization, the default is json.dumps()
- **json_loads** (Callable[[str], Dict[str, Any]]) – Function for JSON deserialization, the default is json.loads()
- **loop** (Optional[AbstractEventLoop]) – Event loop used to schedule tasks. If *loop* is None then asyncio.get_event_loop() is used to get the default event loop.

coroutine open(*self*)

Establish a connection with the CometD server

This method works mostly the same way as the *handshake* method of CometD clients in the reference implementations.

Raises

- ***ClientError*** – If none of the connection types offered by the server are supported

- ***ClientInvalidOperation*** – If the client is already open, or in other words if it isn't *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the handshake or the first connect request gets rejected by the server.

Return type None

coroutine close(*self*)
Disconnect from the CometD server

Return type None

coroutine publish(*self*, *channel*, *data*)
Publish *data* to the given *channel*

Parameters

- **channel** (str) – Name of the channel
- **data** (Dict[str, Any]) – Data to send to the server

Return type Dict[str, Any]

Returns Publish response

Raises

- ***ClientInvalidOperation*** – If the client is *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the publish request gets rejected by the server

coroutine subscribe(*self*, *channel*)
Subscribe to *channel*

Parameters **channel** (str) – Name of the channel

Raises

- ***ClientInvalidOperation*** – If the client is *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the subscribe request gets rejected by the server

Return type None

coroutine unsubscribe(*self*, *channel*)
Unsubscribe from *channel*

Parameters **channel** (str) – Name of the channel

Raises

- ***ClientInvalidOperation*** – If the client is *closed*
- ***TransportError*** – If a network or transport related error occurs
- ***ServerError*** – If the unsubscribe request gets rejected by the server

Return type None

coroutine receive(*self*)
Wait for incoming messages from the server

Return type Dict[str, Any]

Returns Incoming message

Raises

- ***ClientInvalidOperation*** – If the client is closed, and has no more pending incoming messages
- ***ServerError*** – If the client receives a confirmation message which is not `successful`
- ***TransportTimeoutError*** – If the transport can't re-establish connection with the server in `connection_timeout` time.

closed
Marks whether the client is open or closed

Return type `bool`

subscriptions
Set of subscribed channels

Return type `Set[str]`

connection_type
The current connection type in use if the client is open, otherwise `None`

Return type `Optional[ConnectionType]`

pending_count
The number of pending incoming messages

Once *open* is called the client starts listening for messages from the server. The incoming messages are retrieved and stored in an internal queue until they get consumed by calling *receive*.

Return type `int`

has_pending_messages
Marks whether the client has any pending incoming messages

Return type `bool`

4.2.2 ConnectionType

class aiocometd.**ConnectionType**
CometD Connection types

LONG_POLLING = 'long-polling'
Long polling connection type

WEBSOCKET = 'websocket'
Websocket connection type

4.2.3 Extensions

class aiocometd.**Extension**
Bases: `abc.ABC`

Defines operations supported by extensions

coroutine incoming(*self*, *payload*, *headers=None*)
Process incoming *payload* and *headers*

Called just after a payload is received

Parameters

- **payload** (List[Dict[str, Any]]) – List of incoming messages
- **headers** (Optional[Dict[str, str]]) – Headers to send

Return type None

coroutine outgoing(*self*, *payload*, *headers*)
Process outgoing *payload* and *headers*

Called just before a payload is sent

Parameters

- **payload** (List[Dict[str, Any]]) – List of outgoing messages
- **headers** (Dict[str, str]) – Headers to send

Return type None

class aiocometd.**AuthExtension**
Bases: aiocometd.extensions.Extension

Extension with support for authentication

coroutine authenticate(*self*)
Called after a failed authentication attempt

For authentication schemes where the credentials are static it doesn't makes much sense to reimplement this function. However for schemes where the credentials can expire (like OAuth, JWT...) this method can be reimplemented to update those credentials

Return type None

4.2.4 Exceptions

Exception types

Exception hierarchy:

```
AiocometdException
    ClientError
        ClientInvalidOperation
    TransportError
        TransportInvalidOperation
        TransportTimeoutError
        TransportConnectionClosed
    ServerError
```

exception aiocometd.exceptions.**AiocometdException**
Base exception type.

All exceptions of the package inherit from this class.

exception aiocometd.exceptions.**ClientError**
ComtedD client side error

exception aiocometd.exceptions.**ClientInvalidOperation**
The requested operation can't be executed on the current state of the client

exception aiocometd.exceptions.**ServerError**(*message*, *response*)
CometD server side error

If the *response* contains an error field it gets parsed according to the specs

Parameters

- **message** (str) – Error description
- **response** (Optional[Dict[str, Any]]) – Server response message

error
Error field in the *response*

Return type Optional[str]

error_args
Arguments part of the error, message field

Return type Optional[List[str]]

error_code
Error code part of the error code part of the error, message field

Return type Optional[int]

error_message
Description part of the error, message field

Return type Optional[str]

message
Error description

Return type str

response
Server response message

Return type Optional[Dict[str, Any]]

exception aiocometd.exceptions.**TransportConnectionClosed**
The connection unexpectedly closed

exception aiocometd.exceptions.**TransportError**
Error during the transportation of messages

exception aiocometd.exceptions.**TransportInvalidOperation**
The requested operation can't be executed on the current state of the transport

exception aiocometd.exceptions.**TransportTimeoutError**
Transport timeout

4.3 Changelog

4.3.1 0.4.4 (2019-02-26)

- Refactor the websocket transport implementation to use a single connection per client

4.3.2 0.4.3 (2019-02-12)

- Fix reconnection issue on Salesforce Streaming API

4.3.3 0.4.2 (2019-01-15)

- Fix the handling of invalid websocket transport responses
- Fix the handling of failed subscription responses

4.3.4 0.4.1 (2019-01-04)

- Add documentation links

4.3.5 0.4.0 (2019-01-04)

- Add type hints
- Add integration tests

4.3.6 0.3.1 (2018-06-15)

- Fix premature request timeout issue

4.3.7 0.3.0 (2018-05-04)

- Enable the usage of third party JSON libraries
- Fix detection and recovery from network failures

4.3.8 0.2.3 (2018-04-24)

- Fix RST rendering issues

4.3.9 0.2.2 (2018-04-24)

- Fix documentation typos
- Improve examples
- Reorganise documentation

4.3.10 0.2.1 (2018-04-21)

- Add PyPI badge to README

4.3.11 0.2.0 (2018-04-21)

- **Supported transports:**
  - `long-polling`
  - `websocket`
- Automatic reconnection after network failures
- Extensions

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index