AsyncAPY

Release 0.3.3

Jul 05, 2020

Contents:

AsyncAPY is a fully fledged framework to easily deploy asynchronous API endpoints over a custom-made protocol (*AsyncAProto*)

It deals with all the low level I/O stuff, error handling and async calls, exposing a high-level and easy-to-use API designed with simplicity in mind.

Some of its features include:

- **Fully** asynchronous
- Automatic timeout and error handling
- Custom application layer protocol
- Groups: multiple functions can handle the same packet and interact with the remote client
- **Powerful, easy-to-use and high level** API for clients, packets and sessions
- Packets filtering

CHAPTER 1

What's new in AsyncAPY 0.4

- Groups and handlers have been completely reworked, check the docs
- The `AsyncAPY` class has been renamed to `Server` and some breaking changes have been done
- The V1 protocol has been deprecated
- The `AsyncAPY.objects` module has been made private. `Client` and `Packet` objects can be imported from the top-level package
- The code for the server has been polished and improved

1.1 AsyncAPY - Getting started

1.1.1 Installing

Right now, the package is not available on PyPi because some custom dependencies need to be documented and published to the index before AsyncAPY can be succesfully installed via `pip`. In the meanwhile, you can run the following command in your terminal/shell (assuming `pip` and `git` are already installed):

```
python3 -m pip install --user git+https://github.com/intellivoid/AsyncAPY
```

This will install AsyncAPY and its dependencies in your system

Note: On Windows systems, unless you are using PowerShell, you may need to replace `python3` with `py` or `py3` for the commands to work, assuming you added it to `PYTHONPATH` when installed Python.

1.1.2 Hello, world!

Let's write our very first API server with AsyncAPY, an echo server.

An echo server is fairly simple, it always replies with the same request that it got from the client

```
from AsyncAPY import Server

server = Server(addr='0.0.0.0',
                port=1500
               )

async def echo_server(client, packet):
    print(f"Hello world from {client}!")
    print(f"Echoing back {packet}...")
    await client.send(packet)
    await client.close()

server.register_handler(echo_server)    # Register our handler

server.start()    # Start the server
```

Save this script into a file named `example.py`.

What happens if we send a packet to our new, shiny, echo server? Let's try to use the testing client to send a packet to our server: create a new empty file, name it `testclient.py` and paste the following

```
import AsyncAPY.defaultclient


client = defaultclient.Client("0.0.0.0", 1500, tls=False)
enc = 'json'
client.connect()
client.send({"test": 1}, encoding=enc)
response = client.receive_all()
print(response)
```

Now open two terminal windows, run `example.py` and then `testclient.py`, your server output should look like the following:

```
[INFO] 10/02/2020 16:39:35 PM  {Client handler} New session started, UUID is 7fd5fab0-
↪5393-44ec-a75d-fa4f2c7e4562

[DEBUG] 10/02/2020 16:39:35 PM (7fd5fab0-5393-44ec-a75d-fa4f2c7e4562) {Client handler}
↪ Expected stream length is 11

[DEBUG] 10/02/2020 16:39:35 PM (7fd5fab0-5393-44ec-a75d-fa4f2c7e4562) {API Parser}␣
↪Protocol-Version is V2, Content-Encoding is json

[DEBUG] 10/02/2020 16:39:35 PM (7fd5fab0-5393-44ec-a75d-fa4f2c7e4562) {API Parser}␣
↪Checking group 0

[DEBUG] 10/02/2020 16:39:35 PM (7fd5fab0-5393-44ec-a75d-fa4f2c7e4562) {API Parser}␣
↪Calling 'echo_server' in group 0

Hello world from Client(127.0.0.1)!

Echoing back Packet({"test": 1})...

[DEBUG] 10/02/2020 16:39:35 PM (7fd5fab0-5393-44ec-a75d-fa4f2c7e4562) {Response␣
↪handler} Sending response to client

[DEBUG] 10/02/2020 16:39:35 PM (7fd5fab0-5393-44ec-a75d-fa4f2c7e4562) {Response␣
↪Handler} Response sent
```

(continues on next page)

(continued from previous page)

```
[INFO] 10/02/2020 16:39:36 PM (7fd5fab0-5393-44ec-a75d-fa4f2c7e4562) {Client handler}␣
↪The connection was closed
```

while your client output will look like this:

```
b'\x00\x00\x00\r\x16\x00{"test": 1}'
```

As you can see, we got the same JSON encoded packet that we sent!

1.1.3 Filters

You can allow only a subset of packet/client pair to reach your handler, see here to know more.

Filters can be applied to a handler by passing one or more `Filter` objects to the `Server.register_handler()`.

An example of a filtered handler can be found in our dedicated examples section

If you have issues with filters, try reading our FAQ on this topic

1.1.4 Using the decorators

Decorators are a nicer way to add handlers to your server. The line `Server.register_handler()` can be also written as follows:

```
@server.add_handler()
async def your_handler(c, p):
    ...
```

The decorator behaves the same as `Server.register_handler` and take the filter object(s) and the group identifier as optional parameters.

1.1.5 Groups

The way AsyncAPY handles incoming requests has been specifically designed to be simple yet effective.

If you register two or more handlers with conflicting/overlapping filters, only the first one that was registered will be executed.

To handle the same request more than once, you need to register the handler in a different handlers group, like in the following example:

```
from AsyncAPY import Server

server = Server(addr='127.0.0.1', port=1500)


@server.add_handler()
async def echo_server(client, packet):
    print(f"Hello world from {client}!")
    print(f"Echoing back {packet}...")
    await client.send(packet)
    await client.close()
```

(continues on next page)

(continued from previous page)

```
@server.add_handler(group=-1)
async def echo_server_2(client, packet):
    print(f"Hello world from {client} inside a group!")
    print(f"Echoing back {packet}...")
    await client.send(packet)
    # packet.stop_propagation()  # This would prevent the packet from being forwarded
↪to the next handler

server.start()
```

The `group` parameter defaults to 0, the lower this number, the higher will be the position of the handler in the queue. In the example above, `group` equals `-1`, that is lower than `0` and therefore causes that handler to execute first. You could have also set it to 1 (or any other value greather than 0) to make it execute last instead.

1.2 AsyncAPY - Frequently Asked Questions

1.2.1 My console output is not ordered, why?

Don't worry, it's nothing related to how you deployed your server.

The reason why this happens is that AsyncAPY uses trio as asynchronous framework, in order to process multiple requests at the same time. This happens because the order in which coroutines are executed is only partially deterministic, this means that trio's runner function will choose according to some non-predictable conditions when and to which functions give execution control. That's also the reason why AsyncAPY has session ids, to allow users to understand what is what.

1.2.2 Why don't my filter pass?

Well, this can happen for a number of reasons. An example could be if you want to allow 2 specific IP addresses to a handler and pass 2 different `Filters.Ip`: This will never work. Why? Because you're telling AsyncAPY *"I want that only clients that have these 2 IP addresses at the same time can reach this handler"*, which is obviously impossible. Try passing the two IP addresses to the same filter as a list, and check if it works.

In general, it's better to have few `Filter` objects which match all your desired conditions, than many smaller filters, as this also has impacts on performance (More filters == more time spent iterating over them to check them)

1.2.3 The encoding of the responses is wrong!

This can be related to many things, but please note that the encoding of the responses is related to the `Content-Encoding` header that you send to the server, but that it is defined **only once per session**. If you send your first request encoded with json, and the next with ziproto, in the same session, you'll get two JSON encoded responses. If you need two different encodings, start a new session by closing and opening a new connection to the server.

1.2.4 What the hell is ZiProto?

Directly from ZiProto's official repo:

ZiProto is a format for serializing and compressing data [...] ZiProto is designed with the intention to be used for transferring data instead of using something like JSON which can use up more bandwidth when you don't intend to have the data shown to the public or end-user

1.3 AsyncAPY - Code Examples

1.3.1 Filters Examples

Here is an example on how to use AsyncAPY's `Filters` objects

```
from AsyncAPY import Server
from AsyncAPY.filters import Filters

server = Server(addr='0.0.0.0', port=1500)

# This filter will match any digit in the 'foo' field,
# and anything in the 'bar' field, e.g.:
# {"foo": 12355, "bar": "anything"}

@server.add_handler(Filters.Fields(foo='\d+', bar=None))
async def filtered_handler(client, packet):
    print(f"Look at this! {client} sent me {packet}!")
    await client.close()


server.start()
```

You can also use multiple `Filters` objects, by doing the following:

```
from AsyncAPY import Server
from AsyncAPY.filters import Filters

server = Server(addr='0.0.0.0', port=1500)

# This filter will match any digit in the 'foo' field,
# and anything in the 'bar' field, e.g.:
# {"foo": 12355, "bar": "123lmao"}
# Also, only packets coming from localhost (127.0.0.1) and from 151.53.88.15, will␣
↪reach
# this handler


@server.add_handler(Filters.Fields(foo='\d+', bar=None), Filters.Ip(["127.0.0.1",
↪"151.53.88.15"]))
async def filtered_handler(client, packet):
    # code here
    ...


server.start()
```

You can pass as many filters as you want in any order. For a detailed look at filters check their docs.

1.3.2 Groups Examples

```
from AsyncAPY import Server

server = Server(addr='127.0.0.1', port=1500)


@server.add_handler()
async def echo_server(client, packet):
    print(f"Hello world from {client}!")
    print(f"Echoing back {packet}...")
    await client.send(packet)
    await client.close()


@server.add_handler(group=-1)
async def echo_server_2(client, packet):
    print(f"Hello world from {client} inside a group!")
    print(f"Echoing back {packet}...")
    await client.send(packet)
    # packet.stop_propagation()  # This would prevent the packet from being forwarded
↪to the next handler

server.start()
```

1.4 AsyncAPY - The Protocol

1.4.1 The protocol - Why not HTTP?

AsyncAProto is built on top of raw TCP, and now you might be wondering: *"Why not HTTP?"*

I know this *kinda looks like a NIH sindrome*, but when I was building this framework, I realized that HTTP was way too overkill for this purpose, and so I thought that creating a simpler and dedicated application protocol to handle simple packets would have done the thing. And it actually did! (Moreover, HTTP is basically TCP with lots of headers so meh)

1.4.2 The protocol - A simple header system

AsyncAProto has just three headers that must be prepended to the payload in this exact order:

- `Content-Length`: A byte-encoded integer representing the length of the packet (excluding this header itself, but including the next ones). The recommended size is 4 bytes
- `Protocol-Version`: A 1 byte-encoded integer that indicates the protocol version. Will be used in future releases, for now it must be set to 22
- `Content-Encoding`: A 1 byte-encoded integer that can either be 0, for JSON, or 1, for ZiProto. Consider that if the server cannot decode the payload because of an error in the header, the server will reject the packet

1.4.3 The protocol - Supported encodings

AsyncAPY has been designed to deal with JSON and ZiProto encoded payloads, depending on configuration and/or client specifications (ZiProto is highly recommended for internal purposes as it has less overhead than JSON)

A JSON packet with a 4 byte header encoded as a big-endian sequence of bytes (Which is the default), to an AsyncAPY server will look like this:

```
\x00\x00\x00\x10\x16\x01{"foo":  "bar"}
```

and the ZiProto equivalent:

```
\x00\x00\x00\x0b\x16\x01\x81\xa3foo\xa3bar
```

Both the byte order and the header size can be customized, by setting the `AsyncAPY.byteorder` and `AsyncAPY.header_size` parameters, but the ones exposed above are the protocol standards

Warning: Internally, also ZiProto requests are converted into JSON-like data structures, and then converted back to ZiProto before being sent to the client. In order to be valid, then, the request MUST have a key-value structure, and then be encoded in ZiProto

1.4.4 The protocol - Warnings

Warning: Please note, that if an invalid header is prepended to the payload, or no header is provided at all, the packet will be considered as corrupted and it'll be ignored.

AsyncAProto's Default Behaviours:

- If the `Content-Length` header is bigger than `AsyncAPY.header_size` bytes, the server will read only `AsyncAPY.header_size` bytes as the `Content-Length` header, thus resulting in undesired behavior (most likely the server won't be able to read the socket correctly, causing the timeout to expire)
- If the packet is shorter than `AsyncAPY.header_size` bytes, the server will attempt to request more bytes from the client until the packet is at least `AsyncAPY.header_size` bytes long and then proceed normally, or close the connection if the process takes longer than `AsyncAPY.timeout` seconds, whichever occurs first
- If the payload is longer than `Content-Length` bytes, the packet will be truncated to the specified size and the remaining bytes will be read along with the next request (Which is undesirable and likely to cause decoding errors)
- If either the `Content-Encoding` or the `Protocol-Version` headers are not valid, the packet will be rejected
- If both `Content-Encoding` and `Protocol-Version` are correct, but the actual encoding of the payload is different from the specified one, the packet will be rejected
- If the complete stream is shorter than `AsyncAPY.header_size + 5` bytes, which is the minimum size of a packet, the packet will be rejected

Note: AsyncAPY is not meant for users staying connected a long time, as it's an API server framework. The recommended timeout is 60 seconds (default)

Warning: Please also know that the byte order is important and **must be consistent** between the client and the server! The number 24 encoded in big endian is decoded as 6144 if decoded with little endian, the same thing happens with little endian byte sequences being decoded as big endian ones, so be careful!

Note: Just as the server must be able to manage any package fragmentation, the clients must also implement the same strategies discussed above

CHAPTER 2

Indices and tables

- genindex
- modindex
- search