AsyncAPY

Release 0.5.0

Feb 09, 2021

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 0.5

- Minor bug fixes
- Package renamed to *asyncapy*

1.1 API Reference

1.1.1 AsyncAPY's package

The AsyncAProto Client

Warning: This is the client used to connect to AsyncAPY server and is different from the internal object passed to handlers located *here*

class asyncapy.client.**Client**(*byteorder: Optional[str] = 'big'*, *header_size: Optional[int] = 4*, *tls: Optional[bool] = True*, *encoding: Optional[str] = 'json'*, *timeout: Optional[int] = 60*)

Bases: object

Basic Python implementation of an AsyncAProto client

Parameters

- **byteorder** (*str*) – The client's byteorder, can either be "big" or "little". Defaults to "big" (standard)
- **header_size** (*int*) – The size in bytes of the Content-Length header for packets, defaults to 4 (standard)
- **tls** (*optional, bool*) – Instructs the client to use the Transport Layer Security encryption standard for the underlying TCP connection, defaults to True. Recommended for production servers

- **encoding** – The session's payload type, used to encode and decode packets. Defaults to `"json"`, but another valid option is `"ziproto"` (a custom encoding which is more compact, but less flexible and not human-readable, recommended for when the information is not meant to be seen by the general public)
- **timeout** (*int, optional*) – The max. duration in seconds to read the socket before timing out, defaults to 60

connect(*hostname: str*, *port: int*)
Connects to the given hostname and port

Parameters

- **hostname** (*str*) – The address of the server to connect to
- **port** (*int*) – The port of the server to connect to

disconnect()
Closes the underlying TCP socket. No further action is needed client-side as the server will automatically drop its end of the pipe when it notices that the client is gone

receive() → Dict[Any, Any]
Receives a complete AsyncAproto packet and returns the decoded payload

receive_raw() → bytes
Reads the internal socket until an entire AsyncAproto packet is complete and returns the raw packet (including headers). An empty byte string is returned if the socket gets closed abruptly

send(*payload: Union[str, Dict[Any, Any]]*)
Sends the given payload across the underlying TCP connection as a proper AsyncAproto packet

Parameters **payload** (*dict or str, optional*) – The payload to send to the server. Pass either a dictionary object or a valid JSON string

AsyncAPY core modules and resources

class asyncapy.core.**Client**(*address: str*, *server*, *stream: trio.SocketStream*, *session: str*, *encoding: str*)
Bases: object

High-level wrapper around AsyncAproto clients

Parameters

- **address** (*str*) – The client's IP address
- **server** (*Server*) – The server object (created internally), needed to send back packets
- **stream** – The trio socket object associated with the client
- **session** (*str*) – The session_id of the client, defaults to None. Note that, internally, this parameter is replaced with a *Session* object
- **encoding** (*str*) – The client's encoding (which can't change across the session). It can either 'json' or 'ziproto'

close()
Closes the client connection

get_sessions()
Returns all the active sessions associated with the client's IP address

send(*packet*, *close: bool = False*)
Sends the given packet to the client

Parameters

- **packet** (*Packet*) – A *Packet* object
- **close** (*bool, optional*) – If `True`, the connection will be closed right after the packet is sent. Set this to `False` to take full advantage of packet propagation, defaults to `False`

class asyncapy.core.**Handler**(*function: function*, *filters: Optional[List[asyncapy.filters.Filter]] = None*)
Bases: `object`

An object meant for internal use. Every function is wrapped inside a `Handler` object together with its filters

Parameters

- **function** (*FunctionType*) – The asynchronous function, accepting two positional parameters (a *Client* and a *Packet* object)
- **filters** (*Optional[List[Filter]], optional*) – An optional list of *Filter* objects, defaults to `None`

call(**args*)
Calls `self.function` asynchronously, passing `*args` as parameters

check(*client: asyncapy.core.Client*, *packet: asyncapy.core.Packet*)
Iteratively calls the *check* method on `self.filters`. Returns `True` if all filters pass, `False` otherwise

Parameters

- **client** – The client to check for
- **packet** – The packet object to check for

Returns `True` if the handler matches all filters, `False` otherwise

Return type bool

class asyncapy.core.**Packet**(*fields: Union[dict, str, bytes]*, *encoding: str*, *sender: Optional[asyncapy.core.Client] = None*)
Bases: `object`

High-level wrapper around AsyncAProto packets. *Packet* objects behave mostly like Python dictionaries and can be iterated over, used with the `in` operator and support getitem expressions with []

Parameters

- **fields** (*Union[dict, str, bytes]*) – The packet's payload: it can either be a dictionary or valid JSON string (it can also be encoded as bytes)
- **encoding** (*str*) – The payload desired encoding, it can either be `"json"` or `"ziproto"`
- **sender** (*Client*, optional) – This parameter is meant to be initialized internally, and points to the *Client* object that sent the associated payload, defaults to `None`

stop_propagation()
Stops a packet from being propagated, see *StopPropagation*

Raises StopPropagation

class asyncapy.core.**Session**(*session_id: uuid.uuid4*, *client: asyncapy.core.Client*, *date: float*)
Bases: object

A client session

Parameters

- **session_id** – The UUID of the session
- **client** – The client object associated with the current session
- **date** (*float*) – The UNIX Epoch timestamp of when the session was created

close()
Closes the associated client connection

get_client()
Returns the associated client object

Exceptions

exception asyncapy.errors.**StopPropagation**
Bases: Exception

This exception is meant to be raised when *Packet*'s *stop_propagation* method gets called to prevent the dispatcher forwarding the packet to the next handler in the queue.

This is useful when using groups, to learn more click *here* and scroll down to "Groups"

Filters

class asyncapy.filters.**Filter**
Bases: abc.ABC

The standard base for all filters

check(*c*, *p*)
Dummy check method

class asyncapy.filters.**Filters**
Bases: *asyncapy.filters.Filter*

This class implements all the filters in AsyncAPY

class APIFactory(*factory: asyncapy.util.APIKeyFactory*, *field_name: str*)
Bases: *asyncapy.filters.Filter*

A class that wraps around the *APIKeyFactory* class and its children, but as a filter. This filter passes if the provided field name is in the incoming packet and its value is a valid API key inside self.factory

Parameters

- **factory** (*APIKeyFactory*) – The *APIKeyFactory* object
- **field_name** (*str*) – The name of the field to lookup into incoming packets, if the field is not present the filter won't pass

check(*_*, *p*)
Implements the method to check if a filter matches a given packet/client pair
Parameters
- **_** (*Client*) – A client object
- **p** – A packet object

Returns `True` if the filter passed, `False` otherwise
Return type bool

class Fields(***kwargs*)
Bases: *asyncapy.filters.Filter*

Filters fields inside packets. This filter accepts an unlimited number of keyword arguments, that can either be `None`, or a valid regex. In the first case, the filter will match if the payload contains the specified field name, while in the other case the field's value will also be validated against `re.match()`, using the provided parameter as pattern.

Parameters **kwargs** – A list of key-word arguments, which reflects the desired key-value structure of a payload

check(*_*, *p*)
Implements the method to check if a filter matches a given packet/client pair
Parameters
- **_** (*Client*) – A client object
- **p** (*Client*) – A packet object

Returns `True` if the packet's payload follows the given structure, `False` otherwise
Return type bool

class Ip(*ips: Union[List[str], str]*)
Bases: *asyncapy.filters.Filter*

Filters one or more IP addresses, allowing only the ones inside the filter to pass

Note: This filter is dynamic, it can be updated at runtime if assigned to a variable

Parameters **ips** (*Union[List[str], str]*) – An ip or a list of ip addresses

Raises **ValueError** – If the provided ip, or ips, isn't a valid IP address

check(*c*, *_*)
Implements the method to check if a filter matches a given packet/client pair
Parameters
- **c** (*Client*) – A client object
- **_** (*Packet*) – A packet object

Returns `True` if the client's IP is in the filters, `False` otherwise
Return type bool

check(*c*, *p*)
Dummy check method

The AsyncAPY Server

class asyncapy.server.**Server**(*addr: Optional[str] = '127.0.0.1'*, *port: Optional[int] = 8081*, *buf: Optional[int] = 1024*, *logging_level: int = 20*, *console_format: Optional[str] = '[%(levelname)s] %(asctime)s %(message)s'*, *datefmt: Optional[str] = '%d/%m/%Y %H:%M:%S %p'*, *timeout: Optional[int] = 60*, *header_size: int = 4*, *byteorder: str = 'big'*, *config: str = None*, *cfg_parser=None*, *session_limit: int = 0*)
Bases: `object`

AsyncAPY server class

Parameters

- **addr** (*str, optional*) – The address to which the server will bind to, defaults to `'127.0.0.1'`

- **port** (*int, optional*) – The port to which the server will bind to, defaults to 8081
- **buf** (*int, optional*) – The size of the TCP buffer, defaults to 1024
- **logging_level** (*int, optional*) – The logging level for the `logging` module, defaults to 10 (*DEBUG*)
- **console_format** (*str, optional*) – The output formatting string for the `logging` module, defaults to `'[%(levelname)s] %(asctime)s %(message)s'`
- **datefmt** (*str, optional*) – A string for the logging module to format date and time, see the *logging* module for more info, defaults to `'%d/%m/%Y %H:%M:%S %p'`
- **timeout** (*int, optional*) – The timeout (in seconds) that the server will wait before considering a connection as dead and close it, defaults to 60
- **header_size** (*int, optional*) – The size of the `Content-Length` header can be customized. In an environment with small payloads a 2-byte header may be used to reduce overhead, defaults to 4
- **byteorder** (*str, optional*) – The order that the server will follow to read packets. It can either be `'little'` or `'big'`, defaults to `'big'`
- **config** (*str, None, optional*) – The path to the configuration file, defaults to `None` (no config)
- **cfg_parser** (class: `configparser.ConfigParser()`) – If you want to use a custom configparser object, you can specify it here
- **session_limit** (*int, optional*) – Defines how many concurrent sessions a client can instantiate, defaults to 0 (no limit)

add_handler(**filters*, ***kwargs*)
Registers an handler, as a decorator. This does the same of *register_handler*, but in a cleaner way.

Parameters

- **handler** (*FunctionType*) – A function object
- **filters** (*Filter, optional*) – Pass one or more filters to allow only a subset of client/packet pair to reach your handler
- **group** (*int, optional*) – The group id, default to 0

register_handler(*handler*, **filters*, ***kwargs*)
Registers an handler

Parameters

- **handler** (*FunctionType*) – A function object
- **filters** (*Filter, optional*) – Pass one or more filters to allow only a subset of client/packet pair to reach your handler
- **group** (*int, optional*) – The group id, default to 0

run_sync_task(*sync_fn*, **args*, *cancellable=False*, *limiter=None*)
Convert a blocking operation into an async operation using a thread.

This is just a shorthand for `trio.to_thread.run_sync()`, check trio's documentation to learn more

setup()
This method is called when the server is started and it has been thought to be overridden by a custom user defined class to perform pre-startup operations

shutdown()
This method is called when the server shuts down and it has been thought to be overridden by a custom user defined class to perform post-shutdown operations

Note that this method is called only after a proper `KeyboardInterrupt` exception is raised

start()
Starts serving asynchronously on `self.addr:self.port`

Extra utilities

class asyncapy.util.**APIKeyFactory**(*size: int = 32*)
Bases: `object`

Generic class to manage a basic storage of API subscriptions and implementing functionality to issue, revoke and reissue keys. This is a sort of dummy class (keys are stored in memory) and should be subclassed by user-defined classes to implement different storage solutions

Parameters **size** (*int, optional*) – The desired size of the API key. By default it's just a random string, defaults to 32

get(*key: str*) → dict
Returns the associated metadata with the given key, raises `KeyError` if the key doesn't exist

Parameters **key** (*str*) – The API key

Returns The associated metadata with the given API key

Return type str

Raises **KeyError** – If the given key does not exist

issue(*metadata: dict = None*)
Returns a new random API key of `self.size` length, saves it and attaches to it the given metadata

Parameters **metadata** (*dict, optional*) – A dictionary object containing meaningful information that is returned with the *get* method. Defaults to `None`

Returns key The generated API key

Return type str

reissue(*key: str*) → str
Reissues an API key, replacing the old `key` with a new one, keeping the old metadata

Parameters **key** (*str*) – The API key to reissue

Raises **KeyError** – If the given key does not exist

Returns The new API key

revoke(*key: str*)
Revokes an API key, removing its associated data from the internal dictionary

Parameters **key** (*str*) – The API key to revoke

Raises **KeyError** – If the given key does not exist

update(*key: str*, *metadata: dict*)
Updates the associated metadata for the given key with the provided value

Parameters

- **key** (*str*) – The API key to update data for

- **metadata** (*dict*) – A dictionary object containing meaningful information that is returned with the *get* method

Raises **KeyError** – If the given key does not exist

1.2 Getting started

1.2.1 Installing

AsyncAPY can be easily installed via pip:

```
python3 -m pip install --user 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.

1.2.2 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.client


client = client.Client(tls=False, encoding="json")
client.connect("0.0.0.0", 1500)
client.send({"test": 1})
response = client.receive()
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

[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:

```
{"test": 1}'
```

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

1.2.3 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 `register_handler`.

An example of a filtered handler can be found in our dedicated *examples page*.

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

1.2.4 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 takes the filter object(s) and the group identifier as optional parameters.

1.2.5 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()


@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.

1.3 Frequently Asked Questions

1.3.1 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.

1.3.2 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)

1.3.3 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.

1.3.4 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

1.4 Code Examples

1.4.1 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)

```

(continues on next page)

(continued from previous page)

```
# 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=r"\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.

1.4.2 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()
```

1.5 The Protocol

1.5.1 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)

1.5.2 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

1.5.3 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

1.5.4 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

Python Module Index

a

Index