hisock

Release 1.2

SSS-Says-Snek

Dec 17, 2021

CONTENTS:

hisock is a higher-level extension of the *socket* module, with simpler and more efficient usages. It completely removes the hassle of headers, and utilizes decorators to maintain a good code structure.

Note: This project is still currently under development. This is also the first package of the developer, SSS_Says_Snek, so if you find something that needs improvement, submit an issue on the GitHub Repository

---

CHAPTER

ONE

TABLE OF CONTENTS

- genindex
- modindex
- search

1.1 Quickstart

These pages provide you with information to start your hisock journey!

1.1.1 Understanding hisock

First of all, what... **IS** hisock? Well, it's a higher-level extension of the `socket` module, with simpler and more efficient usages, sure, but for beginners, it may be a bit confusing.

What is the socket module?

I've mentioned a lot about a `socket` module in Python a lot, but some of y'all may not know *what* socket is. Basically, it's a pretty low-level networking interface that uses "sockets" to communicate between computers over a network. The problem is, it's a bit overwhelming when you start learning sockets.

So, I developed `hisock`, which basically simplifies `socket` down, and provides additional features.

What are the advantages of using hisock over socket?

That's a good question. While `hisock` is still under development, it aims to simplify or eliminate some complex parts of the standard `socket`. For example, hisock uses decorators to simplify code structure, and eliminates the hassle of worrying about headers.

Note: Again, some of you may not know what a header is. When you send data, it is not interpreted as a "message"; instead of messages, there is a "stream" of data, and the client/server decides how many bytes to read from the stream. This creates a problem; how do we know how much of a message to read? This is where headers come in. They are basically data the specifies the length of a "message". In order for this to work, headers **MUST** be fixed-length, so it is usually padded with spaces.

Let's say that I decided that my header length would be 16 bytes long. When a client sends me some data, it will have that header in front, then the actual content. I would first receive the first 16 bytes, and see that it is the number "12",

followed by 14 spaces. At this point, I know that the "message" is 12 bytes long. So, I receive another 12 bytes, to get the message "Hello World!"

How do you send and received data with hisock?

Data sent with `hisock` usually has a *command* before the data (but before the header, of course). Once the command is sent, it can be received using decorators `hisock.server.on` and `hisock.client.on`. An argument will be passed in those decorators, and

1.1.2 Installation

To install hisock, you will need to make sure you have PIP and Python3.7 or later installed and added to your PATH (ensure `pip` and `python` works). There are two main ways to install HiSock. You can install it through PyPi (recommended) or you can install it from the GitHub.

Warning: hisock is the first project I ever published to PyPI, so there might be some quirks on PyPI here and there, like the sudden burst of version post-releases. However, I will try to keep this at the bare minimum, and hopefully figure out PyPI good enough

Installing via PyPI

The recommended way to install a stable version of hisock is through PyPI. To install hisock, open a terminal/command prompt and type:

```
$ python -m pip install hisock (Windows)
    OR
$ pip install hisock (Windows)
    OR
$ python3 -m pip install hisock (Mac/Linux)
    OR
$ pip3 install hisock (Mac/Linux)
```

You've now successfully installed a stable version of hisock!

Installing via GitHub

Sometimes, however, you might want to install the *latest* version of hisock, not just the *stable* version. To do this, you can either download the repository from GitHub here or you can clone the repository (recommended) via Git.

To install from Git, open a terminal/command prompt and type:

```
$ git clone https://github.com/SSS-Says-Snek/hisock.git (Git)
    OR
$ gh repo clone SSS-Says-Snek/hisock (GitHub CLI)
```

Or, you can go onto GitHub and download the the ZIP file by pressing the green "Code" button, then clicking "Download ZIP". You can then unzip the file into a folder with your favorite unzip tool.

Then, go back to the terminal or command prompt and type:

```
$ cd hisock
```

After you're now in the working directory of the repo, install it in editable mode with:

```
$ pip install -e .
```

You should now have successfully installed the latest version of hisock! If this doesn't work, then try one of the alternatives in Method 1, but replace `hisock` with `-e .` (E.g `python3 -m pip install -e .`)

Note: If you want to check if hisock is *actually* installed, run this command in your terminal or command prompt:

```
$ python -c $'try:\n\timport hisock;print(f"Hisock {hisock.__version__} successfully␣
↪installed")\nexcept Exception as e:print(f"Failed to install hisock for {e} reason")'
```

1.1.3 Tutorial

In this "tutorial", I'm going to be explaining the basic parts of `hisock`, and how to use them to create working programs.

It is also highly encouraged that you read the *Understanding Hisock* before reading the tutorial.

> **Caution:** I might explain some things wrong, so if you see something wrong with my explanation, I encourage you to try to submit a pull request on Github to fix it!

Note: Due to certain reasons, for the client part, I will be referring the server's IP as `utils.get_local_ip()`, the same IP address as where the server is hosted at. Of course, in real applications, there will either be input for the server IP, or a hardcoded server IP

First of all, we need to install hisock. Assuming you have `pip`, you can install hisock with either `python -m pip install hisock` for Windows, or `pip3 install hisock` for Mac/Linux. Refer to *the installation guide* for more details.

First of all, network structures are divided into *servers* and *clients*. Servers basically *serve* data, and can also be a mean of communicating between clients. Clients are just the users that give and receive the server data.

Note: For example, if you are creating a multiplayer game, you can create a *server* that *clients* need to join; once they do, client `A` can ask the server to share some info to client `B`, and so on and so forth.

Creating our first server

There is a function to create a server in `hisock`, it is `hisock.start_server()`. To host a server, we'd need a tuple with two arguments; the first argument is the IP address of the server. Hisock provides a way to find your local IP address, with `hisock.utils.get_local_ip()`. The second argument is the *port* of the server. I won't go into ports, but usually a number between 1024 and 65535 would suffice.

Hisock servers and clients have a `run()` and `update()` method respectively, in order to run correctly. However, you need to run them in a while loop, as to minimize lag time. So, our final server script looks like:

```
from hisock import start_server, utils

server = start_server((utils.get_local_ip(), 36969))  # Haha funny

while True:
    server.run()
```

That's basically it! Of course, this server is useless, but hey, it's a step in the right direction! We'll add on to this later on.

Creating our first client

Obviously, without a client, a server's kind of pointless. So, let's spice things up, with some boilerplate client code!

Now, the first thing we need to do, is to connect to the server. We can do that with hisock's `connect()` function. Like `start_server()`, it takes one tuple as an argument, with two elements; The first one is the IP of the server, and the second one is the port of the server.

Also like `HiSockServer`, `HiSockClient` needs to be ran in a while loop. However, in `HiSockClient`, instead of the `run()` method, it is called `update()`. So, our final boilerplate client code is:

```
from hisock import connect, utils

# This is a bit tricky - This will only work on the same computer
# running the server, as it gets the same IP (unless you port forward)
client = connect((utils.get_local_ip()), 36969)

while True:
    client.update()
```

Like the server, this doesn't do anything at all yet, but soon, we'll finally add some functionality to the server and client!

Clearing some things up

Let me clear some stuff up first; data sent with `hisock` usually has a *command* before the data. Once the command is sent, it can be received using decorators `hisock.server.on` and `hisock.client.on`. An argument will be passed in those decorators that specifies the command to listen data with those commands. When data with the command attached is found, `hisock` will call that function, and pass in a few arguments regarding message content (`hisock.server` will also have an argument about the client data).

Let's start with a decorator example for the server

```
# Server
server = ...

@server.on("random_command")
def random_cmd_handler(clt_data, message):
    # clt_data is a dict of client information
    # message is the data content, in bytes

    print(message)
```

If any data is found with the command "random_command" attached before it, then it will call `random_cmd_handler()`, filling in the parameters with the appropriate values.

Finally, we have an example of the client

```
client = ...

@client.on("another_random_command")
def handler_thing(message):
    # No clt_data, as server always sends message
    print(message)
```

This isn't much different to the server; any data that has the command "another_random_command" attached to it, will automatically call `handler_thing()`, albeit with less parameters

Now that we've done that, let's add functionality to our bland server and client!

Adding (some) functionality to our server

So far, we have made a server and client, but it doesn't really *do* anything. So, it's time to add some functionality, starting with the server!

Now, let's say that we want to print the client's IP on the server side, whenever the server connects to a client. `hisock` provides something I like to call "reserved functions", where there are certain commands that get attached to data that occur on very special events. For server, there are a few, including:

1. `join` occurs whenever a client connects
2. `leave` occurs whenever a client disconnects
3. `message` occurs whever a client sends a "message"

(I mean, they're pretty self-explanatory)

Anyways, we can use the `join` reserved function to print the client's IP, like so:

```
# Server
from hisock import iptup_to_str
...
server = ...

@server.on("join")
def clt_join(clt_data):  # Of course, no message on join
    print(
        f"Cool, {iptup_to_str(clt_data['ip'])} joined!"
    )  # the IP is stored in a tuple, with a (str IP, int Port) format

while True:
    server.run()
```

Now, if we run the client on this updated server, we will see the IP address of the client!

Of course, this is still not that interesting on the client side, so we'll finally start to send some data in the next part!

Sending data to our client

Obviously, if we don't have a way of sending data, there isn't any use of hisock. `hisock.server` provides the `.send_client()`, `.send_all_clients()`, and `.send_client_raw()` methods to send data to a specific client. **With the exception of** `send_client_raw()`, the methods usually need the client to send to, command to associate the data, and the data itself.

Note: Right, I've mentioned about *commands* a lot in this tutorial, but haven't really explained what it is. To clean up code structure, hisock divides the data receiving part with decorators; refer to *Clearing some things up* for more details.

Anyways, we got our organized data receiving, but now, how do we actually receive the data? Well, `hisock` data **usually** have a command before them, so that hisock can know which data should be sent to which function (as you will see later on, the commands on data **aren't** required)

We will be discussing more in-depth about what `send_all_clients()` and `send_client_raw()` does, but we shall focus on `send_client()` for now

---

So, about `send_client()`: This method of `HiSockServer` is used to... send data to a specific client. It accepts 3 arguments: the client (we'll be using its IP in this case), the command, and the data. The client's IP can either be in the form "IP.IP.IP.IP:Port" as a string, **OR** as a two-element tuple, like ("IP.IP.IP.IP", Port). We'll be using the latter one in this case.

Remember: **The data must either be a bytes-like object (E.g b"sussy"), or a dictionary (E.g {"sus": "amogus"})**

Let's say that we as soon as a client joins, the server should pick a random integer from 1 to 10000, and send it back to the client. This is perfectly doable, and is pretty straightforward! Our server code would be:

```
# Server
import random
...
server = ...

@server.on("join")
def clt_join(clt_data):  # Of course, no message on join
    print(
        f"Cool, {iptup_to_str(clt_data['ip'])} joined!"
    )  # the IP is stored in a tuple, with a (str IP, int Port) format
    randnum = random.randint(1, 10000)
    server.send_client(clt_data['ip'], "random", str(randnum).encode())

...
```

While we sent the data to the client, the client still has no way of interpreting this new data! So, we must modify our client

```
# Client
client = ...

@client.on("random")
def interpret_randnum(msg):
    randnum = int(msg)
    print(f"Random number generated by the server is a {randnum}!")

...
```

Now, whenever the client joins that server, it will receive the data sent by it! How cool is that?

Sending data to our server

By common sense, HiSockClients provide a way to send data to the server, with `send()` and `raw_send()`. Again, **with the exception of** `raw_send()`, the send methods accept two arguments; the first being the command of the data, and the second being the data itself.

Like `HiSockServer`, **The data must be a bytes-like object or a dictionary**

Now, let's say that after the client got its random number, we want to send to ther server a message saying, hey, we received it, good for you. We could edit our client like:

```
# Client
client = ...

@client.on("random")
def interpret_randnum(msg):
    randnum = int(msg)
    print(f"Random number generated by the server is a {randnum}!")
    client.send("verif", b"I GOT IT")

...
```

and our server can be

```
# Server
server = ...

@server.on("join")
def clt_join(clt_data):  # Of course, no message on join
    print(
        f"Cool, {iptup_to_str(clt_data['ip'])} joined!"
    )  # the IP is stored in a tuple, with a (str IP, int Port) format
    randnum = random.randint(1, 10000)
    server.send_client(clt_data['ip'], "random", str(randnum).encode())

@server.on("verif")
def verif_msg(clt_data, message):
    print(f"Successfully sent the number to {iptup_to_str(clt_data['ip'])}!")

...
```

We've successfully made a functional client and server!

Conclusion

This wraps up the basics of `hisock`, but there is a lot more to know! If you are interested, I highly recommend you to follow the **Intermediate Tutorial** (Still not created yet kek), where I'll be covering some less beginner-friendly features of `hisock`. See you soon!

Note: While you *can* create some basic applications with some basic knowledge of `hisock`, but for larger, more robust applications, it is not recommended, but **necessary** to have a better understanding of it.

Refer to the *Tutorials Page* for more tutorials

1.1.4 Examples

Examples are located in the *./examples* directory in hisock. You can take a look at some of them. The basic client-server examples are:

Client

```
"""
Basic example of the structure of `hisock`. This is the client script.
Not an advanced example, but gets the main advantages of hisock across
"""

import time

from hisock import connect, get_local_ip

server_to_connect = input("Enter server IP to connect to (Press enter for default of␣
↪your local IP): ")
port = input("Enter Server port number (Press enter for default of 6969): ")

if server_to_connect == '':
    server_to_connect = get_local_ip()

if port == '':
    port = 6969
else:
    port = int(port)

name = input("Name? (Press enter for no name) ")
group = input("Group? (Press enter for no group) ")

print("======================================== ESTABLISHING CONNECTION␣
↪========================================")

if name == '':
    name = None
if group == '':
    group = None

join_time = time.time()
```

(continues on next page)

(continued from previous page)

```
s = connect(
    (server_to_connect, port),
    name=name, group=group)

@s.on("hello_message")
def handle_hello(msg):
    print("Thanks, server, for sending a hello, just for me!")
    print(f"Looks like, the message was sent on timestamp {msg.decode()}, "
          f"which is just {round(float(msg.decode()) - join_time, 6) * 1000}␣
↪milliseconds since the connection!")
    print("In response, I'm going to send the server a request to do some processing")

    s.send("processing1", b"randnum**2")
    result = int(s.recv_raw())

    print(f"WHOAAA! The result is {result}! Thanks server!")

while True:
    s.update()
```

Server

```
import sys
import time
import random

from hisock import start_server, get_local_ip, iptup_to_str

ADDR = get_local_ip()
PORT = 6969

if len(sys.argv) == 2:
    ADDR = sys.argv[1]
if len(sys.argv) == 3:
    PORT = int(sys.argv[2])

print(f"Serving at {ADDR}")
server = start_server((ADDR, PORT))

@server.on("join")
def client_join(client_data):
    print(f"Cool, {':'.join(map(str, client_data['ip']))} joined!")
    if client_data['name'] is not None:
        print(f"    - With a sick name \"{client_data['name']}\", very cool!")
    if client_data['group'] is not None:
        print(f"    - In a sick group \"{client_data['group']}\", cool!")

    print("I'm gonna send them a quick hello message")

    server.send_client(client_data['ip'], "hello_message", str(time.time()).encode())
```

(continues on next page)

(continued from previous page)

```
@server.on("processing1")
def process(client, process_request: str):
    print(f"\nAlright, looks like {iptup_to_str(client['ip'])} received the hello␣
↪message, "
          "\nas now they're trying to compute something on the server, because they have
↪"
          "potato computers")
    print("Their processing request is:", process_request)

    for _ in range(process_request.count("randnum")):
        randnum = str(random.randint(1, 100000000))
        process_request = process_request.replace("randnum", randnum, 1)

    result = eval(process_request)  # Insecure, but I'm lazy, so...
    print(f"Cool! The result is {result}! I'mma send it to the client")
    server.send_client_raw(client['ip'], str(result).encode())


while True:
    server.run()
```

1.2 More Tutorials

These tutorials are aimed at people who have just finished the Quickstart Tutorial, but want to learn more about `hisock`. As I've stated in the Quickstart tutorial, it is not recommended, but necessary, to have a deeper understanding of `hisock` to create more complex applications.

1.2.1 Intermediate-level Tutorial

In this "tutorial", I'll talk about some more advanced topics that I *may* have mentioned in the quickstart, but never talked about it. These topics are extremely important in order to develop a complex `hisock` application.

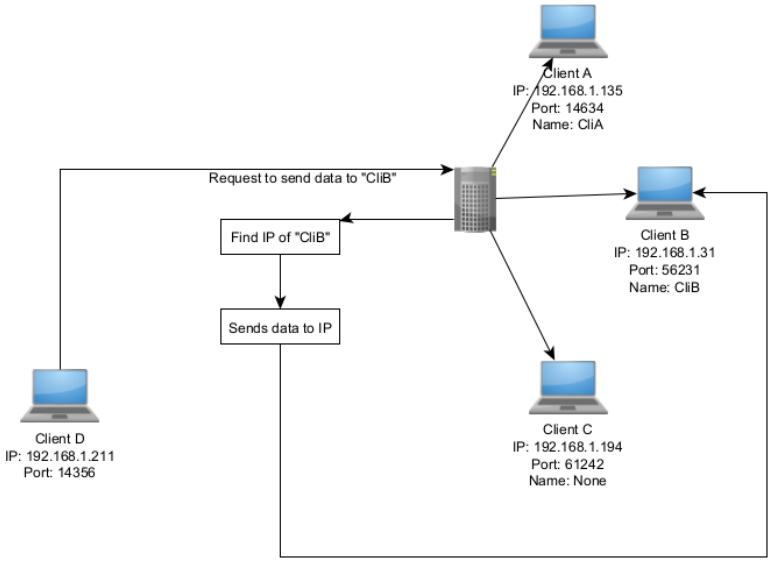
Without further ado, let's start!

Where we left off

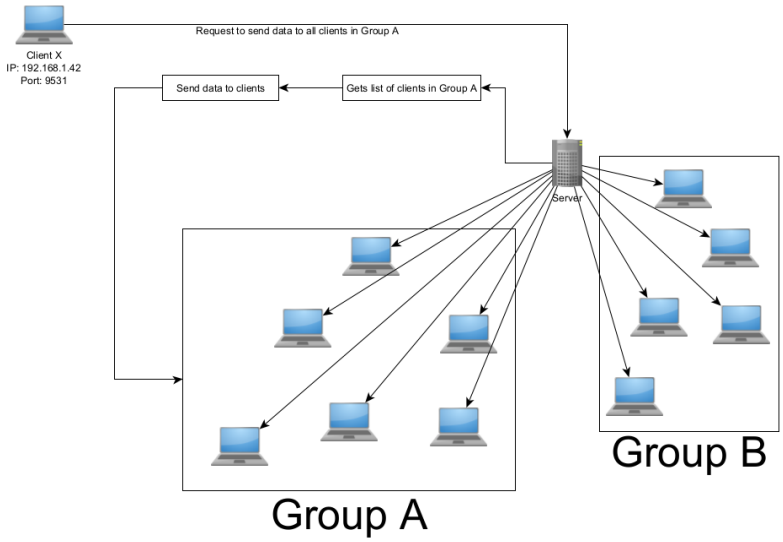
In the quickstart tutorial, we left off knowing how to send data from server to client, and client to server. While there is nothing much to the client-to-server interaction, with only one method `.send()`, there are **FIVE** methods for a server-to-client interaction; *send_client()*, *send_all_clients()*, *send_client_raw()*, *send_group()*, and *send_group_raw()*. While we have covered the most obvious one (`send_client()`), we have not yet covered the other two.

Names and groups

Now, I'll clear up some more stuff before we move on. Hisock provides what I like to call **names** and **groups**. Under-the-hood, `hisock` usually identifies clients and servers by their **IP Address**. This works most of the times, but sometimes, you want to differentiate between clients, *without* knowing the IP. This is where names come in; On client connection, you can pass a `name` argument into `connect()`, as to bind a name to the client. Now, using some additional functions, we could send and receive data by using the client name!



Now, on to groups. Like names, groups are another way of identifying a client, but instead of **one** client, it can identify multiple! With groups, you can organize clients by what they correspond to. For example, if you are making a multiplayer game, and you are making some sort of lobby for a limited number of clients, then a group would be a good way to identify which clients are in which lobby.



Note: I've actually never used groups before, which is pretty... uh... sad. I added groups because I thought of some use cases for it, but I never bothered to use it in an example, so... yeah.

Now, let's finally dig into the `send` methods!

The other `send` methods

In the beginner tutorial, we've covered the *send_client()* method that allows the server to send data to specific clients. But, what if we wanted to send data to all the clients? Well, that is exactly what *send_all_clients()* does. Yes, the name is pretty self-explanatory.

Now, let's move on to *send_client_raw()*! Sometimes, while you do want to send data to a client, you don't necessarily want to send it **with a command**. For example, if a server just sent out information to a client, and a client sent back some more information, it would be much easier to directly send data to the client, instead of sending a command, and needing to make another function. So, *send_client_raw()* allows you to send data *without a command*.

Remember when I said that groups can be used to organize clients? Well, how do we communicate and send data to a group? As you may have guessed by the name, *send_group()* sends data to a specific group, taking a group name and the data as parameters.

1.3 API Reference

1.3.1 hisock.server

A module containing the main server classes and functions, including `HiSockServer` and `start_server()`

Note: Header lengths usually should not be that long; on average, header lengths are about 64 bytes long, which is more than enough for most cases (10 vigintillion, 10**64). Plus, a release is planned for hisock that utilizes ints to bump the header utilization from *10**x* to *2**(7x)* (where x is the header length)

---

class hisock.server.**HiSockServer**(*addr: tuple[str, int]*, *blocking: bool = True*, *max_connections: int = 0*, *header_len: int = 16*, *cache_size: int = - 1*, *tls: Union[dict, str] = None*)

The server class for hisock HiSockServer offers a neater way to send and receive data than sockets. You don't need to worry about headers now, yay!

Parameters

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number of where the server should be hosted. Due to the nature of reserved ports, it is recommended to host the server with a port number that's higher than 1023. Only IPv4 currently supported
- **blocking** (*bool, optional*) – A boolean, set to whether the server should block the loop while waiting for message or not. Default passed in by *start_server()* is True
- **max_connections** (*int, optional*) – The number of maximum connections *HiSockServer* should accept, before refusing clients' connections. Pass in 0 for unlimited connections. Default passed in by *start_server()* is 0
- **header_len** (*int, optional*) – An integer, defining the header length of every message. A smaller header length would mean a smaller maximum message length (about 10**header_len). Any client connecting MUST have the same header length as the server, or else it will crash. Default passed in by *start_server()* is 16 (maximum length: 10 quadrillion bytes)

Variables

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number

- **header_len** (*int*) – An integer, storing the header length of each "message"
- **clients** (*dict*) – A dictionary, with the socket as its key, and the client info as its value
- **clients_rev** (*dict*) – A dictionary, with the client info as its key, and the socket as its value
- **funcs** (*dict*) – A list of functions registered with decorator *on()*. **This is mainly used for under-the-hood-code**

Note: It is advised to use *get_client()* or *get_all_clients()* instead of using `clients` and `clients_rev`

Also, **only IPv4 is currently supported**

---

close()
Closes the server; ALL clients will be disconnected, then the server socket will be closed.

Running *server.run()* won't do anything now. :return:

disconnect_all_clients(*force=False*)
Disconnect all clients.

disconnect_client(*client: str*)
Disconnects a specific client Different formats of the client is supported. It can be:

- An IP + Port format, written as "ip:port"
- A client name, if it exists

Parameters **client** – The client to disconnect. The format could be either by IP+Port, or a client name

get_addr() → tuple[str, int]
Gets the address of where the hisock server is serving at.

Returns A tuple, with the format (str IP, int port)

get_all_clients(*key: Union[Callable, str] = None*) → list[dict[str, str]]
Get all clients currently connected to the server. This is recommended over the class attribute *self._clients* or *self.clients_rev*, as it is in a dictionary-like format

Parameters **key** (*Union[Callable, str], optional*) – If specified, there are two outcomes: If it is a string, it will search for the dictionary for the key, and output it to a list (currently support "ip", "name", "group"). Finally, if it is a callable, it will try to integrate the callable into the output (CURRENTLY NOT SUPPORTED YET)

Returns A list of dictionaries, with the clients

Return type list[dict, ...]

get_client(*client: Union[str, tuple[str, int]]*) → dict[str, Union[str, socket.socket]]
Gets a specific client's information, based on either:

1. The client name
2. The client IP+Port
3. The client IP+Port, in a 2-element tuple

Parameters **client** (*Union[str, tuple]*) – A parameter, representing the specific client to look up. As shown above, it can either be represented as a string, or as a tuple.

Raises

- **ValueError** – Client argument is invalid
- **TypeError** – Client does not exist

Returns A dictionary of the client's info, including IP+Port, Name, Group, and Socket

Return type dict

get_group(*group: str*) → list[dict[str, Union[str, socket.socket]]]
Gets all clients from a specific group

Parameters **group** (*str*) – A string, representing the group to look up

Raises **TypeError** – Group does not exist

Returns A list of dictionaries of clients in that group, containing the address, name, group, and socket

Return type list

on(*command: str*, *threaded: bool = False*) → Callable
A decorator that adds a function that gets called when the server receives a matching command

Reserved functions are functions that get activated on specific events. Currently, there are 3 for HiSock-Server:

1. join - Activated when a client connects to the server
2. leave - Activated when a client disconnects from the server
3. message - Activated when a client messages to the server

The parameters of the function depend on the command to listen. For example, reserved commands *join* and *leave* have only one client parameter passed, while reserved command *message* has two: Client Data, and Message. Other nonreserved functions will also be passed in the same parameters as *message*

In addition, certain type casting is available to nonreserved functions. That means, that, using type hints, you can automatically convert between needed instances. The type casting currently supports:

1. bytes -> int (Will raise exception if bytes is not numerical)
2. bytes -> str (Will raise exception if there's a unicode error)

Type casting for reserved commands is scheduled to be implemented, and is currently being worked on.

Parameters

- **command** (*str*) – A string, representing the command the function should activate when receiving it
- **threaded** – A boolean, representing if the function should be run in a thread in order to not block the run() loop.

  Defaults to False

Returns The same function (The decorator just appended the function to a stack)

Return type function

run()
Runs the server. This method handles the sending and receiving of data, so it should be run once every iteration of a while loop, as to not lose valuable information

Note: This is the main method to run HiSockServer. This **MUST** be called every iteration in a while loop, as to keep waiting time as short as possible between client and server. It is also recommended to put this in a thread.

send_all_clients(*command: str*, *content: Union[bytes, dict[Union[str, int, float, bool, None], Union[str, int, float, bool, None]]]*)

Sends the commmand and content to *ALL* clients connected

Parameters

- **command** (*str*) – A string, representing the command to send to every client
- **content** (*Union[bytes, dict]*) – A bytes-like object, containing the message/content to send to each client

send_client(*client: Union[str, tuple[str, int]]*, *command: str*, *content: Union[bytes, dict[Union[str, int, float, bool, None], Union[str, int, float, bool, None]]]*)

Sends data to a specific client. Different formats of the client is supported. It can be:

- An IP + Port format, written as "ip:port"
- A client name, if it exists
- A tuple with an (IP, Port) format

Parameters

- **client** (*Union[str, tuple]*) – The client to send data to. The format could be either by IP+Port, or a client name
- **command** (*str*) – A string, containing the command to send
- **content** (*Union[bytes, dict]*) – A bytes-like object, with the content/message to send

Raises

- **ValueError** – Client format is wrong
- **TypeError** – Client does not exist
- **UserWarning** – Using client name, and more than one client with the same name is detected

send_client_raw(*client*, *content: bytes*)

Sends data to a specific client, *without a command* Different formats of the client is supported. It can be:

- An IP + Port format, written as "ip:port"
- A client name, if it exists
- A tuple with an (IP, Port) format

Parameters

- **client** (*Union[str, tuple]*) – The client to send data to. The format could be either by IP+Port, or a client name
- **content** (*Union[bytes, dict]*) – A bytes-like object, with the content/message to send

Raises

- **ValueError** – Client format is wrong
- **TypeError** – Client does not exist

- **Warning** – Using client name and more than one client with the same name is detected

send_group(*group: str*, *command: str*, *content: Union[bytes, dict[Union[str, int, float, bool, None], Union[str, int, float, bool, None]]]*)

Sends data to a specific group. Groups are recommended for more complicated servers or multipurpose servers, as it allows clients to be divided, which allows clients to be sent different data for different purposes.

Parameters

- **group** (*str*) – A string, representing the group to send data to
- **command** (*str*) – A string, containing the command to send
- **content** (*Union[bytes, dict]*) – A bytes-like object, with the content/message to send

Raises **TypeError** – The group does not exist

send_group_raw(*group: str*, *content: bytes*)

Sends data to a specific group, without commands. Groups are recommended for more complicated servers or multipurpose servers, as it allows clients to be divided, which allows clients to be sent different data for different purposes.

Non-command-attached content is recommended to be used alongside with `HiSockClient.recv_raw()`

Parameters

- **group** (*str*) – A string, representing the group to send data to
- **content** (*bytes*) – A bytes-like object, with the content/message to send

Raises **TypeError** – The group does not exist

class hisock.server.**ThreadedHiSockServer**(*addr*, *blocking=True*, *max_connections=0*, *header_len=16*)

A downside of *HiSockServer* is that you need to constantly *run()* it in a while loop, which may block the program. Fortunately, in Python, you can use threads to do two different things at once. Using *ThreadedHiSockServer*, you would be able to run another blocking program, without ever fearing about blocking and all that stuff.

Note: In some cases though, *HiSockServer* offers more control than *ThreadedHiSockServer*, so be careful about when to use *ThreadedHiSockServer* over *HiSockServer*

---

join()

Waits for the thread to be killed

run()

The main while loop to run the thread

Refer to *HiSockServer* for more details

> **Warning:** This method is **NOT** recommended to be used in an actual production enviroment. This is used internally for the thread, and should not be interacted with the user

start_server()

Starts the main server loop

stop_server()

Stops the server

hisock.server.**start_server**(*addr*, *blocking=True*, *max_connections=0*, *header_len=16*)

Creates a *HiSockServer* instance. See *HiSockServer* for more details

Returns A *HiSockServer* instance

hisock.server.**start_threaded_server**(*addr*, *blocking=True*, *max_connections=0*, *header_len=16*)
Creates a *ThreadedHiSockServer* instance. See *ThreadedHiSockServer* for more details

Returns A *ThreadedHiSockServer* instance

1.3.2 hisock.client

A module containing the main client classes and functions, including `HiSockClient` and `connect()`

class hisock.client.**HiSockClient**(*addr: tuple[str, int]*, *name: Union[str, None]*, *group: Union[str, None]*, *blocking: bool = True*, *header_len: int = 16*, *cache_size: int = - 1*)
The client class for hisock. HiSockClient offers a higher-level version of sockets. No need to worry about headers now, yay! HiSockClient also utilizes decorators to receive messages, as an easy way of organizing your code structure (methods are provided, like *recv_raw()*, of course)

Parameters

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number of the server wishing to connect to Only IPv4 currently supported
- **name** (*str, optional*) – Either a string or NoneType, representing the name the client goes by. Having a name provides an easy interface of sending data to a specific client and identifying clients. It is therefore highly recommended to pass in a name

  Pass in NoneType for no name (*connect* should handle that)
- **group** (*str, optional*) – Either a string or NoneType, representing the group the client is in. Being in a group provides an easy interface of sending data to multiple specific clients, and identifying multiple clients. It is highly recommended to provide a group for complex servers

  Pass in NoneType for no group (*connect* should handle that)
- **blocking** (*bool, optional*) – A boolean, set to whether the client should block the loop while waiting for message or not. Default sets to True
- **header_len** (*int, optional*) – An integer, defining the header length of every message. A smaller header length would mean a smaller maximum message length (about 10**header_len). The header length MUST be the same as the server connecting, or it will crash (hard to debug though). Default sets to 16 (maximum length of content: 10 quadrillion bytes)

Variables

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number
- **header_len** (*int*) – An integer, storing the header length of each "message"
- **name** (*str*) – A string, representing the name of the client to identify by. Defaults to None
- **group** (*str*) – A string, representing the group of the client to identify by. Defaults to None
- **funcs** (*dict*) – A list of functions registered with decorator *on()*.

> **Warning:** This is mainly used for under-the-hood-code, so it is **NOT** recommended to be used in production-ready code

change_group(*new_group: Optional[str]*)
Changes the client's group

Parameters **new_group** (*Union[str, None]*) – The new group name of the client

change_name(*new_name: Optional[str]*)
Changes the name of the client

Parameters **new_name** (*Union[str, None]*) – The new name for the client to be called

close(*emit_leave: bool = True*)
Closes the client; running *client.update()* won't do anything now

Parameters **emit_leave** (*bool*) – Decides if the client will emit *leave* to the server or not

on(*command: str*, *threaded: bool = False*) → Callable
A decorator that adds a function that gets called when the client receives a matching command

Reserved functions are functions that get activated on specific events. Currently, there are 2 for HiSockClient:

1. client_connect - Activated when a client connects to the server
2. client_disconnect - Activated when a client disconnects from the server

The parameters of the function depend on the command to listen. For example, reserved functions *client_connect* and *client_disconnect* gets the client's data passed in as an argument. All other nonreserved functions get the message passed in.

In addition, certain type casting is available to nonreserved functions. That means, that, using type hints, you can automatically convert between needed instances. The type casting currently supports:

1. bytes -> int (Will raise exception if bytes is not numerical)
2. bytes -> float (Will raise exception if bytes is not numerical)
3. bytes -> str (Will raise exception if there's a unicode error)

Type casting for reserved commands is scheduled to be implemented, and is currently being worked on.

Parameters

- **command** (*str*) – A string, representing the command the function should activate when receiving it
- **threaded** (*bool, optional*) – A boolean, representing if the function should be run in a thread in order to not block the update() loop.

  Defaults to False

Returns The same function (The decorator just appended the function to a stack

Return type function

raw_send(*content: bytes*)
Sends a message to the server: NO COMMAND REQUIRED. This is preferable in some situations, where clients need to send multiple data over the server, without overcomplicating it with commands

Parameters **content** (*bytes*) – A bytes-like object, with the content/message to send

recv_raw() → bytes
Waits (blocks) until a message is sent, and returns that message. This is not recommended for content with commands attached; it is meant to be used alongside with `HiSockServer.send_client_raw()` and `HiSockServer.send_group_raw()`

Returns A bytes-like object, containing the content/message the client first receives

Return type bytes

send(*command: str*, *content: Union[bytes, dict[Union[str, int, float, bool, None], Union[str, int, float, bool, None]]]*)
Sends a command & content to the server, where it will be interpreted

Parameters

- **command** (*str*) – A string, containing the command to send
- **content** (*Union[bytes, dict]*) – A bytes-like object, with the content/message to send

update()
Handles newly received messages, excluding the received messages for *wait_recv* This method must be called every iteration of a while loop, as to not lose valuable info. In some cases, it is recommended to run this in a thread, as to not block the program

---

Note: This is the main method to run HiSockClient. This **MUST** be called every iteration in a while loop, as to keep waiting time as short as possible between client and server. It is also recommended to put this in a thread.

---

class hisock.client.**ThreadedHiSockClient**(*addr*, *name=None*, *group=None*, *blocking=True*, *header_len=16*, *cache_size=- 1*)
A downside of *HiSockClient* is that you need to constantly *run()* it in a while loop, which may block the program. Fortunately, in Python, you can use threads to do two different things at once. Using *ThreadedHiSockClient*, you would be able to run another blocking program, without ever fearing about blocking and all that stuff.

Note: In some cases though, *HiSockClient* offers more control than *ThreadedHiSockClient*, so be careful about when to use *ThreadedHiSockClient* over *HiSockClient*

---

join()
Waits for the thread to be killed

run()
The main while loop to run the thread

Refer to *HiSockClient* for more details

> **Warning:** This method is **NOT** recommended to be used in an actual production enviroment. This is used internally for the thread, and should not be interacted with the user

start_client()
Starts the main server loop

stop_client()
Stops the client

hisock.client.**connect**(*addr*, *name=None*, *group=None*, *blocking=True*, *header_len=16*, *cache_size=- 1*)
Creates a *HiSockClient* instance. See HiSockClient for more details

Parameters

- **addr** (*tuple*) – A two-element tuple, containing the IP address and the port number
- **name** (*str, optional*) – A string, containing the name of what the client should go by. This argument is optional

- **group** (*str, optional*) – A string, containing the "group" the client is in. Groups can be utilized to send specific messages to them only. This argument is optional
- **blocking** (*bool, optional*) – A boolean, specifying if the client should block or not in the socket.

  Defaults to True
- **header_len** (*int, optional*) – An integer, defining the header length of every message.

  Defaults to True

Returns A *HiSockClient* instance

Return type instance

hisock.client.**threaded_connect**(*addr*, *name=None*, *group=None*, *blocking=True*, *header_len=16*, *cache_size=- 1*)

Creates a *ThreadedHiSockClient* instance. See *ThreadedHiSockClient* for more details

Returns A *ThreadedHiSockClient* instance

1.3.3 hisock.utils

A module containing some utilities to either:

1. Help `hisock.client` and `hisock.server` run (denoted with leading underscore)
2. Provide functions to use alongside hisock

hisock.utils.**get_local_ip**(*all_ips: bool = False*) → str

Gets the local IP of your device, with sockets

Parameters **all_ips** (*bool, optional*) – A boolean, specifying to return all the local IPs or not. If set to False (the default), it will return the local IP first found by `socket.gethostbyname()`

Default: False

Returns A string containing the IP address, in the format "ip:port"

Return type str

hisock.utils.**input_client_config**(*ip_prompt: str = 'Enter the IP of the server: '*, *port_prompt: str = 'Enter the Port of the server: '*, *name_prompt: Union[str, None] = 'Enter name to connect as: '*, *group_prompt: Union[str, None] = 'Enter group to connect to: '*) → tuple[Union[str, int], ...]

Provides a built-in way to obtain the IP and port of the configuration of the server to connect to

Parameters

- **ip_prompt** (*str, optional*) – A string, specifying the prompt to show when asking for IP.

  Default is "Enter the IP of the server: "
- **port_prompt** (*str, optional*) – A string, specifying the prompt to show when asking for Port

  Default is "Enter the Port of the server: "
- **name_prompt** (*Union[str, None], optional*) – A string, specifying the prompt to show when asking for Client Name

  Default is "Enter name to connect as: " (Pass in None for no input)

- **group_prompt** (*Union[str, None], optional*) – A string, specifying the prompt to show when askign for Client Group

  Default is "Enter group to connect to: " (Pass in None for no input)

Returns A tuple containing the config options of the server

Return type tuple[str, int, Optional[str], Optional[int]]

hisock.utils.**input_server_config**(*ip_prompt: str = 'Enter the IP of where to host the server: '*, *port_prompt: str = 'Enter the Port of where to host the server: '*) → tuple[str, int]

Provides a built-in way to obtain the IP and port of where the server should be hosted, through `input()`

Parameters

- **ip_prompt** (*str, optional*) – A string, specifying the prompt to show when asking for IP.

  Default is "Enter the IP of where to host the server: "

- **port_prompt** (*str, optional*) – A string, specifying the prompt to show when asking for Port

  Default is "Enter the Port of where to host the server: "

Returns A two-element tuple, consisting of IP and Port

Return type tuple[str, int]

hisock.utils.**ipstr_to_tup**(*formatted_ip: str*) → tuple[str, int]

Converts a string IP address into a tuple equivalent

Parameters **formatted_ip** (*str*) – A string, representing the IP address.

Must be in the format "ip:port"

Returns A tuple, with IP address as the first element, and an INTEGER port as the second element

Return type tuple[str, int]

hisock.utils.**iptup_to_str**(*formatted_tuple: tuple[str, int]*) → str

Converts a tuple IP address into a string equivalent

This function is like the opposite of `ipstr_to_tup`

Parameters **formatted_tuple** (*tuple[str, int]*) – A two-element tuple, containing the IP address and the port. Must be in the format (ip: str, port: int)

Returns A string, with the format "ip:port"

Return type str

1.4 Changelog

This page keeps track of all the new features that were added to, modified, or removed in specific versions. This may be useful for selecting which version is best for you, if the latest version does not work for you.

1.4.1 v1.1

New features

- A message cache for HiSockClient! The message cache allows you to view the last x messages sent by the client. Configuration of the message cache is available on creation of the instance.
- `.disconnect_client()`, `.disconnect_all_clients()`, and `.close()` methods for HiSockServer, as well as the `force_disconnect` reserved function for HiSockClient! It's now easier to force disconnect of a client from a server with these functions. The `force_disconnect` reserved function will be triggered when the client has been disconnected from the server with these functions.
- Add list and dict type casts for HiSockClient.

Improved features

- Documentation improvements, yay!
- Handle hisock errors better (not crashes!). New exceptions have been created for specific causes of server/client errors.
- cleancode.py can now clean up the code, with the help of black! So, the code has been PEP8-ified, for our (my) avid PEP8 fans.
- Move HiSockClient and HiSockServer to be available on import of hisock.
- Move __version__ to be available on import of hisock.
- Separate requirements.txt into that and requirements_contrib.txt. Some of the requirements are actually just requirements for the tests to run.

Bug fixes

- (Maybe) fix bug for the .close() method of HiSockClient.
- Fixed a fatal import error of hisock.

Other

Hisock now has a new logo, created by sheepy0125! Hisock also has a discord server! However, we're (I'm) still setting it up, so the invite isn't public yet.

1.4.2 v1.0

New features

- New examples have been added to hisock! Now, you can play a Tic-Tac-Toe game made in hisock! There is also the addition of the example shown in the README
- HiSockClient, HiSockServer, and their threaded counterparts now support some dunder methods!
- More type casts have been added
- Ability to change name and group after client initialization has been added (change_name() and change_group())

- A built-in way to obtain server and client configuration through inputs has been added (input_server_config() and input_client_config())

Improved features

- Of course, we have some documentation improvements!
- Pictures are starting to appear in the documentation
- I added a beginner's tutorial to get started on hisock
- I am currently working on another intermediate hisock tutorial, which covers the more advanced topics
- A new changelog page has been added
- Python docstrings have been improved
- Hisock error handling has been improved again
- Added classifiers in PyPI

Bug fixes

- The regular expression used in 0.1 and earlier has been replaced with a newer one, in order to respond against certain edge cases
- Bumped cryptography module from 3.4.8 to 35.0.0 (security patches)

1.4.3 v0.1

This version is the first minor version of `hisock`! It contains several major code accessibility things added, though not a lot of in-usage code has been added in this version.

New features

- PyPI installation of hisock (`python -m pip install hisock` or `pip3 install hisock`)
- Documentation for `hisock`, hosted here! (ReadTheDocs)
- New support for threaded `HiSockServer` and `HiSockClient`

Improved features

- Better traceback handling

1.4.4 v0.0.1 (GRAND RELEASE)

The first version of hisock! Contains all the basics of what hisock can do, including:

- Name/Group feature to identify specific clients
- Decorators to handle message receiving
- Under-the-hood code that handles headers
- Type-casting receiving function arguments

INDEX