DocName

Release 1.0

YourName

Nov 26, 2022

CONTENTS

CHAPTER ONE

SPHINX_DOCUMENTATION

1.1 client module

This file is the file on the client's side of things. It handles all the actions of the client, including sending and receiving messages, logging to and exiting from the servers and all the other functions.

`client.`**`IP`**

The IP address of the client

Type
str

`client.`**`PORT`**

The port of the client

Type
int

`client.`**`PORT_S`**

The port of the server that it is connecting to

Type
int

`client.`**`IP_S`**

The IP of the server that it is connecting to

Type
str

`client.`**`PORT_BALANCER`**

The port of the load balancer

Type
int

`client.`**`MAX_SIZE`**

The max size limit for the message

Type
int

`class` `client.`**`Client`**`(`*`*args`*`)`

Bases: `object`

Class for the client to communicate with the server

`Receive`(*key*)

This function handles all the data that is received by the client

Parameters
`key` (*obj*) – An object which holds the socket through which the communication happens, along with other information

`Send`(*key*)

This is the function that deals with the sending of messages. There are different options for direct messages, group emssages, creation, addition and deletion from a group.

Parameters
`key` (*obj*) – An object which contains the socket through which the data is to be sent, and will store the data that is to be sent through the socket

Returns
False if the socket closes and the user disconnects, *True* otherwise

Return type
bool

`call_balancer`()

This function calls the load balancer, which changes the port numbers of the servers to which it connects to. It then connects to the specific server, and disconnects the previous connection with the load balancer from the clients end.

`handle`()

This is the function that handles the different possibilities of events. It handles read and write events, as well as the initial connecting of the client to the server.

Threading has been used to separate the input and the output processes, to make sure that we can send input while not blocking the receiving of output.

`init_connect`()

This is the function that initially connects to the server. This is before the client is registered or authenticated :returns: *True* if the process :rtype: bool

Raises
`SystemExit` – If the user wants to exit

`login`(*key*)

This function is for when a client is logging in, when the client has already been registered before

Parameters
`key` (*obj*) – Contains information about the socket and data inside it

Returns
True if successful login, *False* if not

Return type
bool

Raises
`SystemExit` – If user wants to exit

`message_read`()

Reads from the selector and calls the *Receive()* function if the event is to read

Returns
0 when the exit flag is 1 (which never happens)

Return type
int

`message_send()`

Reads from the selector and calls the *Send()* function if the event is to write

Returns
0 when the user wants to exit, it keeps running until the user presses 0

Return type
int

`register`(*key*)

The function that registers to the server, when the user is appearing for the first time

Parameters
key (*obj*) – Contains information about the socket through which the communication is happening between the server and the client

Returns
False if any invalid things are entered, *True* if everything goes properly

Return type
bool

Raises
SystemExit – If the user wants to exit

1.2 database module

This file is a collection of functions to access and modify the postgreSQL database *fastchat*.

We have used the package psycopg2, which provides us a connector through which we are able to query, modify and change the database.

`class database.CentralDatabase`

Bases: `object`

Class for the Central Database, to be accessed by all the servers

`add_participant`(*part_id*, *admin_id*, *group_id*)

Adds a participant from a group, on the admin's request

Parameters

- **part_id** (*int*) – The ID of the participant to be added
- **admin_id** (*int*) – The ID of the person trying to add the participant
- **group_id** (*_type_*) – The group to which the participant is to be added

Returns
-1, if the person trying to add a participant is not the admin of the group -2, if the person to be added isn't part of the group -3, if the group doesn't exist -4, if the user to be added doesn't exist 1, otherwise

Return type
int

`change_server`(*user_id*, *server_id*)

Change the server of the user in the users table

Parameters

- **`user_id`** (*`int`*) – The user ID of the user
- **`server_id`** (*`int`*) – The server ID to be updated to

`check_cred`(*user_id*, *password*)

Checks whether the credentials supplied by the user are correct

Parameters

- **`user_id`** (*`int`*) – The user ID of the user
- **`password`** (*`str`*) – The password (encrypted) of the user

Returns

True if the credentials are valid and *False* if the credentials aren't

Return type

bool

`check_server`(*user_id*)

summary

Parameters

`user_id` (*`int`*) – The user ID of the user

Returns

The server that the user is conencted to, -1 if offline and -2 if not registered

Return type

int

`check_user`(*user_id*)

Checks whether a user is registered or not

Parameters

`user_id` (*`int`*) – The user ID of the user

Returns

True if the user is registered and *False* if not

Return type

bool

`close_connection`()

Closes the connection between the python script and the postgreSQL database

`create_group`(*admin_id*, *participants*)

Creates a group with an admin and participants

Parameters

- **`admin_id`** (*`int`*) – The admin user ID
- **`participants`** (`list` of `int`) – The list of participants to be added in the group

Returns

Returns the ID of the group created, -1 if a participant in the list isn't registered

Return type

int

`del_participant`(*part_id*, *admin_id*, *group_id*)

Deletes a participant from a group, on the admin's request

Parameters

- `part_id` (*int*) – The ID of the participant to be removed
- `admin_id` (*int*) – The ID of the person trying to remove the participant
- `group_id` (*_type_*) – The group from which the participant is to be removed

Returns

-1, if the person trying to remove a participant is not the admin of the group -2, if the person to be removed isn't part of the group -3, if the group doesn't exist 1, otherwise

Return type

int

`displayallgroupmessage`()

Print all the unread group messages

`displayallmessage`()

Displays all the messages stored in the tables messages

`displayallusers`()

Prints the details of all users

`fetch_group_keys`(*group_id*, *user_id*)

Gets the public keys of all users in a group except the given user (Checks whether they're in the group or not)

Parameters

- `group_id` (*int*) – The group ID of the group
- `user_id` (*int*) – The user ID of the user to be excluded

Returns

A dictionary with keys being the user IDs and values being the public keys. Returns -1 if the user isn't present in the group

Return type

dict or *int*

`fetch_key`(*user_id*)

Gets the public key of a given user

Parameters

`user_id` (*int*) – Contains the user ID of the user

Returns

Public Key of the user

Return type

str

`get_min_numclients`()

Selects the server with the least number of clients connected to it

Returns

Server ID of the required server

Return type
int

`group_participants`(*group_id*)

Gets all the participants of a group

Parameters
`group_id` (*int*) – The group ID of the group

Returns
List of participant IDs in the group

Return type
`list` of `int`

`init_numclients`(*num_servers*)

Initializes the table numclients which maintains the tally of number of clients per server vs the corresponding ip and port number. Sets all values to 0

Parameters
`num_servers` (*int*) – The number of servers running

Returns
True if it initialises correctly, else *False*

Return type
bool

`insert_group_message`(*sender_id*, *receiver_id*, *datetime*, *group_id*, *message*)

Inserts an group message into the groupmessage table, only when unread by a receiver

Parameters

- `sender_id` (*int*) – The user ID of the sender
- `receiver_id` (*int*) – The user ID of the receiver who is offline
- `datetime` (*str*) – The datetime string of the message
- `group_id` (*int*) – The user ID of the sender
- `message` (*str*) – The unread group message to the offline receiver

Returns
True if inserting works and *False* otherwise

Return type
bool

`insert_message`(*sender_id*, *receiver_id*, *datetime*, *message*)

Inserts a direct message into the message database, only when the receiver is offline

Parameters

- `sender_id` (*int*) – The user ID of the sender
- `receiver_id` (*int*) – The receiver ID of the user
- `datetime` (*str*) – The datetime string of the message
- `message` (*_type_*) – The direct text message to be sent

Returns
True if the receiver is registered and *False* if not

Return type
bool

`insert_newuser`(*user_id*, *password*, *server_id*, *public_key*)

Inserts a new user into the user table, if the user did not exist previously

Parameters

- **`user_id`** (*`int`*) – The user ID of the user to be inserted
- **`password`** (*`str`*) – The encrypted password of the user
- **`server_id`** (*`int`*) – The server ID of the server that the user is connected to
- **`public_key`** (*`str`*) – The public key of the user

Returns
True if the user wasn't already present, and *False* if it was already present

Return type
bool

`show_group_message`(*receiver_id*)

Returns the unread group messages from the groupmessages table, with the given receiver ID. Also deletes those messages from the table

Parameters
`receiver_id` (*`int`*) – The user ID of the receiver who is offline

Returns
Returns the row of the database with the correct receiver ID

Return type
`list` of `tup`

`show_message`(*receiver_id*)

Returns the unread messages from the messages table, with the given receiver ID. Also deletes those messages from the table

Parameters
`receiver_id` (*`int`*) – The user ID of the receiver who is offline

Returns
Returns the row of the database with the correct receiver ID

Return type
`list` of `tup`

`update_numclients`(*server_ID*, *increase*)

Updates the table numclients when user comes online or goes offline

Parameters

- **`server_ID`** (*`int`*) – ID of the server to be updated
- **`increase`** (*`int`*) – Either +1 or -1 depending on whether a user is going offline or coming online

Returns
True if there is no error, else *False* if the server itself isn't in the table to begin with

Return type
bool

`database.`**`list_to_postgre_array`**`(`*lst*`)`

Converts a Python list into a PostgreSQL array

Parameters
`lst` (*`list`*) – Contains the list elements to be converted into a PostgreSQL array

Returns
A PostgreSQL array in string format

Return type
str

`database.`**`postgre_array_to_list`**`(`*arr_str*`)`

Converts a PostgreSQL array into a Python list

Parameters
`arr_str` (*`str`*) – PostgreSQL array in string format

Returns
List of elements in the PostgreSQL array

Return type
list

1.3 enc module

This module helps in performing End to End encryption in the entire project. We use a combination of RSA (asymmetric encryption) and (symmetric key block cipher) to encrypt the messages. This modules has two classes AESCipher to perform AES encryption and Encryption class, which perfoms E2EE of messages.

Message is encrypted using AES and the AES key is encrypted using RSA.

`class` `enc.`**`AESCipher`**`(`*key*`)`

Bases: `object`

Parameters
`key` (*`bytes or str`*) – This key is used to encrypt/decrypt the messages.

Return type
AESCipher object.

`decrypt``(`*enc*`)`

Parameters
`enc` (*`bytes or str`*) – Base 64 encoded encrypted message that is to be decrypted.

Returns
Decrypted message.

Return type
bytes

`encrypt``(`*raw*`)`

Parameters
`raw` (*`str or bytes`*) – The message that is to be encrypted.

Returns
Base 64 encoded encrypted message.

Return type
bytes

class enc.**Encrypt**(**args*)

Bases: `object`

Parameters
***args** (*(optional)*) – len(args) = 0 : creates Encrypt object with new RSA Private-Public key pair. len(args) = 1 : (str) creates Encrypt object with RSA Private-Public keys loaded from args[0]_public.pem and args[0]_private.pem

RSA_decrypt(*message*)

RSA decrypts using self.private_key

Parameters
message (*bytes or str*) – Message that is to be decrypted.

Returns
Decrypted message if successful, else returns None.

Return type
bytes or None

RSA_encrypt(*message*, *key*)

RSA encrypts using the PEM format public key provided.

Parameters

- **message** (*bytes or str*) – message that is to be RSA encrypted.
- **key** (*bytes*) – DESCRIPTION.

Returns
Encrypted message.

Return type
bytes

RSA_sign(*message*)

Signs the message using self.private_key

Parameters
message (*str or bytes*) – message that is to be signed.

Returns
Base 64 encoded sign.

Return type
bytes

RSA_verify(*message*, *sign*, **args*)

Verifies the signature with the message.

Parameters

- **message** (*str or bytes*) – message that is to be verified.
- **sign** (*bytes*) – Base 64 encoded signature.
- ***args** (*optional*) –

Returns
True if verified, False if not.

Return type
bool

`decrypt`(*encrypted_message*, *encrypted_key*)

Parameters

- **`encrypted_message`** (*`bytes`*) – encrypted message.
- **`encrypted_key`** – base 64 encoded encrypted AES key.

Returns
message – decrypted message.

Return type
bytes

`encrypt`(*message*, *key*)

Encrypts the message with given PEM format public key.

Parameters

- **`message`** (*`str or bytes`*) – message that is to be encrypted.
- **`key`** (*`str or bytes`*) – PEM format public key.

Returns

- **encrypted_key** (*bytes*) – base 64 encoded encrypted AES key.
- **encrypted_message** (*bytes*) – encrypted message.
- **key** (*str or bytes*) – PEM format public key.

`get_public_key`(**args*)

Return/store RSA public key.

If no argument is given function returns the public key.

If one argument is given, it must be str, private key is stored in the file args[0]_public.pem

Parameters
`*args` (*`(optional)`*) –

Returns
if no argument is given

Return type
bytes

`load_keys`(*file*)

Loads RSA public and private keys from ‘file_public.pem’ and ‘file_private.pem’

Parameters
`file` (*`str`*) – file name.

Return type
None.

`return_public_key_object`(*public_key*)

Creates RSA.PublicKey object corresponding to the PEM format public key.

Parameters
`public_key` (*`bytes`*) – PEM format public key.

Returns
RSA.PublicKey object.

Return type
RSA.PublicKey

`save_keys`(*file*)

Saves RSA public and private keys to the files 'file_public.pem' and 'file_private.pem' respectively.

Parameters
`file` (*`str`*) – file name.

Return type
None.

`save_private_key`(**args*)

Return/store RSA private key.

If no argument is given function returns the private key.

If one argument is given, it must be str, private key is stored in the file args[0]_private.pem

Parameters
`*args` (*`(optional)`*) –

Returns
if no argument is given

Return type
bytes

1.4 loadbalancer module

This file is the code for the load balancer, where clients first connect to. The load balancer has two algorithms, one being a simple round robin method to choose between servers and the other being a method that finds out the load of each server, by checking the number of connections to each server and sending the client to the one with the least number.

`loadbalancer.L_IP`

The IP address of the load balancer

Type
str

`loadbalancer.L_PORT`

The port of the load balancer

Type
int

`loadbalancer.START_PORT`

The starting port of all the servers, the servers will be connected to consecutive ports starting from the *START_PORT*

Type
int

`loadbalancer.`**`NUM_SERVERS`**

The total number of servers running

Type
str

`loadbalancer.`**`SERVER_POOL`**

A list of tuples containing the IP address and the PORTS of all the servers

Type
list

`loadbalancer.`**`ITER`**

A cycle of the *SERVER_POOL* list

Type
cycle

`loadbalancer.`**`MAX_SIZE`**

The max size limit for the message

Type
int

`class loadbalancer.`**`loadbalancer`**`(`*IP*, *PORT*, *algorithm*`)`

Bases: `object`

This class is for the load balancer, and specifies the connections between the load balancer and the servers, the load balancing strategy and accepting and sending messages to the clients

`accept_connection``()`

This function accepts a connection from a client and then forwards it to a particular server (based on the algorithm) with a sign, to verify that it was in fact the load balancer that sent him to that server

`close_conn``(`*sock*`)`

This function closes the connection from the load balancer between the load balancer and the client

Parameters
`key` (*`obj`*) – Contains information about the socket between the client and the server, and also the data that will be communicated through that socket

`handle``()`

This function handles the events regarding the load balancer through a selector, by handling the read and write events separately

`min_conn``()`

Queries the database for the server with the least number of connections

Returns
Returns a tuple of IP, PORT for the server to connect to, in this case, it is the one with the least number of connections

Return type
tuple

`round_robin``(`*iter*`)`

Picks the next server in the cycle of servers

Parameters
`iter` (*`iterable`*) – The last tuple that had gotten assigned

Returns
Returns a tuple of IP, PORT for the server to connect to, in this case, it is the next in the cycle of servers

Return type
tuple

`select_server`(*server_list*, *algorithm*)

This determines which algorithm to use in order to select the server for the client to go to.

Parameters

- **`server_list`** (*`list`*) – The list of servers that are running
- **`algorithm`** (*`str`*) – A string with the algorithm name

Returns
Returns a tuple of IP, PORT for the server to connect to

Return type
tuple

Raises
`Exception` – When the algorithm is not recognised by the load balancer

1.5 server module

This file is the file on the server's side of things. It handles all the actions of the server, including sending and receiving messages, communicating with other servers, looking at the database, etc

`server.IP`

The IP address of the server

Type
str

`server.PORT`

The port of the server

Type
int

`server.ID`

The server ID of the server (should be in sequence, the first server to be created must have ID 1, and so on)

Type
int

`server.N`

The number of servers that are to be created

Type
int

`server.MAX_SIZE`

The max size limit for the message

Type
int

`class server.Server(IP, PORT, ID, N)`

Bases: `object`

`accept_wrapper(sock)`

This function accepts the initial condition from the client, before it is authenticated by the server. It also calls the function to update the numclients table, which gives us an idea on the load of the servers

Parameters

sock (*obj*) – The socket used for communication between client and server, before authentication

`add_to_group(part_id, admin_id, group_id, key)`

This is a function for user *admin_id* to add user *part_id* into the group *group_id*

Parameters

- **part_id** (*int*) – The ID of the participant to be added
- **admin_id** (*int*) – The ID of the person who is adding the participant
- **group_id** (*int*) – The ID of the group to be added
- **key** (*obj*) – Contains information about the socket between the client and the server, and also the data that will be communicated through that socket. In this case, we use it to forward a confirmation message about the addition into the group

`authenticate_client(key, mask)`

This function authenticates the client if it has already been registered, with it's password and ID

Parameters

- **key** (*obj*) – Contains information about the socket between the client and the server, and also the data that will be communicated through that socket
- **mask** (*obj*) – Gives information about what type of operation must be performed, whether it is a read operation or a write operation

Raises

KeyError – If the username and password don't match the records

`bind_listener(*args)`

This binds the listening socket to the port and IP of the server, and sets the setblocking parameter to *False*

`connect_servers()`

This function establishes connections between each pair of servers

`create_group(key, details)`

Create a group with the list of participants and the admin of the group to be formed

Parameters

- **key** (*obj*) – Contains information about the socket through which the request came from, so can be used to find the admin of the new group.
- **details** (*_type_*) – The message, which contains information on the new participants of the group.

`forward(key)`

This function forwards the messages that have been generated with both a read and write status to the client directly, and then updates the socket back to an empty message

Parameters
`key` (*obj*) – Contains information about the socket between the client and the server, and also the data that will be communicated through that socket

`get_group_keys`(*details*, *user_id*)

Get a dictionary containing key value pairs of all members of a group and their public keys

Parameters

- `details` (*dict*) – The message attribute of this dict contains the group id of the group
- `user_id` (*int*) – The user id to be excluded from the dict (he's querying for the other keys)

Returns
dict with key value pairs of the participants of the group and their public keys (except the one of the user_id)

Return type
dict

`get_keys`(*details*)

Get a dictionary containing a single key value pair of the user and his public key

Parameters
`details` (*dict*) – The message attribute of this dict contains the user id of the user

Returns
dict with a single key value pair of the user and his public key

Return type
dict

`group_chat`(*key*, *details*)

This function is responsible for sending group chat messages between two clients, by handling messages between servers and also between clients and servers

Parameters

- `key` (*obj*) – Contains information about the socket between the client and the server, and also the data that will be communicated through that socket
- `details` (*dict*) – A JSON object which contains the encrypted message to be sent to members of the group

`handle_events`()

This is the function that handles the different events that can occur. It first reads from the selector, and then depending on the event, it decides to receive or send messages.

`message`(*key*)

This is a function to accept a message from a client, and then handle the following events appropriately, like which server to redirect the message to, etc. It also calls the group chat functions

Parameters
`key` (*obj*) – Contains information about the socket between the client and the server, and also the data that will be communicated through that socket.

`register_client`(*key*, *details*)

This function takes care of the registration of the client (for the first time only, after that it is authentication)

Parameters

- `key` (*obj*) – Contains information about the socket between the client and the server, and also the data that will be communicated through that socket

- `details` (*dict*) – Is a JSON object which has the information about the message sent by the client to the server, namely, the user and the encrypted password

`remove_from_group`(*part_id*, *admin_id*, *group_id*, *key*)

This is a function for user *admin_id* to remove user *part_id* from the group *group_id*

Parameters

- `part_id` (*int*) – The ID of the participant to be removed
- `admin_id` (*int*) – The ID of the person who is removing the participant
- `group_id` (*int*) – The ID of the group to be removed
- `key` (*obj*) – Contains information about the socket between the client and the server, and also the data that will be communicated through that socket. In this case, we use it to forward a confirmation message about the removal from the group

`reply`(*key*)

This function is used to only send messages to the client, and modify the socket to it's next state, which could either be a read state or both read and write state. This function also is responsible for querying from the table for unread messages, which would then be delivered to the respective clients

Parameters

`key` (*obj*) – Contains information about the socket between the client and the server, and also the data that will be communicated through that socket

`server_messages`()

This function is used to send messages between servers, since all the servers are connected to each other. This is essential to be able to send messages between two clients who are not connected to the same server

`service_connection`(*key*, *mask*)

This function handles events that are only about receiving messages (or images) from clients or servers. There is a separate section to deal with messages that are about authentication

Parameters

- `key` (*obj*) – Contains information about the socket between the client and the server, and also the data that will be communicated through that socket.
- `mask` (*obj*) – Contains information about the operation to be performed, whether it is to receive data or to send data

PYTHON MODULE INDEX