31.03.2022

Individual Portfolio Assignment 1

DATA 2410: Network and cloud systems

Bernt Moritz Schmid Olsen (s341528)
OSLOMET

Table of content

Introduction 2
Using the solution 2
  Requirements 2
  Start the solution 3
  Using the client application 5
  Using the server application 5
Chatbots 7
Explanation of the server program 10
  The simple application layer protocol 13
Results 14
Conclusion 16
References 17

Introduction

This document contains a report for the Individual portfolio assignment in the course "Datanettverk og skytjenester". It contains a description of how the code created for this assignment can be run, how the code works and the results it generates. The code is available on Github at this address:

https://github.com/Oslo-Metropolitan-University-OsloMet/individual-portfolio-assignment-1-B-Moritz

The solution to this assignment consists of the two python applications server.py and client.py. The server.py program runs a simple chat server on the end point running the program. The client.py program starts a client application which connects several bots to the simple chat server. The bots reply to conversation initiators sent by the host. This is thereby simulating a single threaded chat service with users connected to the server.

The solution also contains a third python module, YrInterface.py, which is used to connect to the Met API in order to obtain weather data for a certain location. This module is only used by the weatherbot running in the client.py application and it is depending on that the client has access to the internet.

Using the solution

Requirements

There are some things that need to be considered before starting the server and the client application. First the server.py and client.py files must be downloaded to their end points. It is not a problem to run the server and client on the same end system.

The server.py file should be in the same directory as the conversationInitiators.txt file. This file is available in the repository uploaded to GitHub. It contains the messages that the host is sending to the clients in order to start conversations. Running the server.py program would work without this file since there is a list of messages written directly in the code which is used if the file is not found.

There is also added some logging capabilities to the server program. It will generate new log files each day and store them in the "Logs" folder. The "Logs" folder should be in the same folder as the server.py file. The folder is created automatically if it does not already exist.

The client.py program also has some requirements to the code environment. The following files should be in the same directory as the client.py file:

worldcities.csv - A csv file containing data about world cities. The data is provided by simplemaps and can be downloaded from their website: https://simplemaps.com/data/world-cities. The file used is the free csv file licensed with the Creative Commons license. The file is also added to the repository on GitHub.

YrInterface.py – The module used to retrieve weather data from the Met API (Metreologisk institutt): https://developer.yr.no/

In order to satisfy some of the requirements of the Met API, the data received must be cached for a period. The YrInterface.py module is therefore saving weather data to a file for every location (city). The files are stored in the "WeatherCache" folder. This folder should be in the same folder as the client.py file and is created automatically if it does not exist.

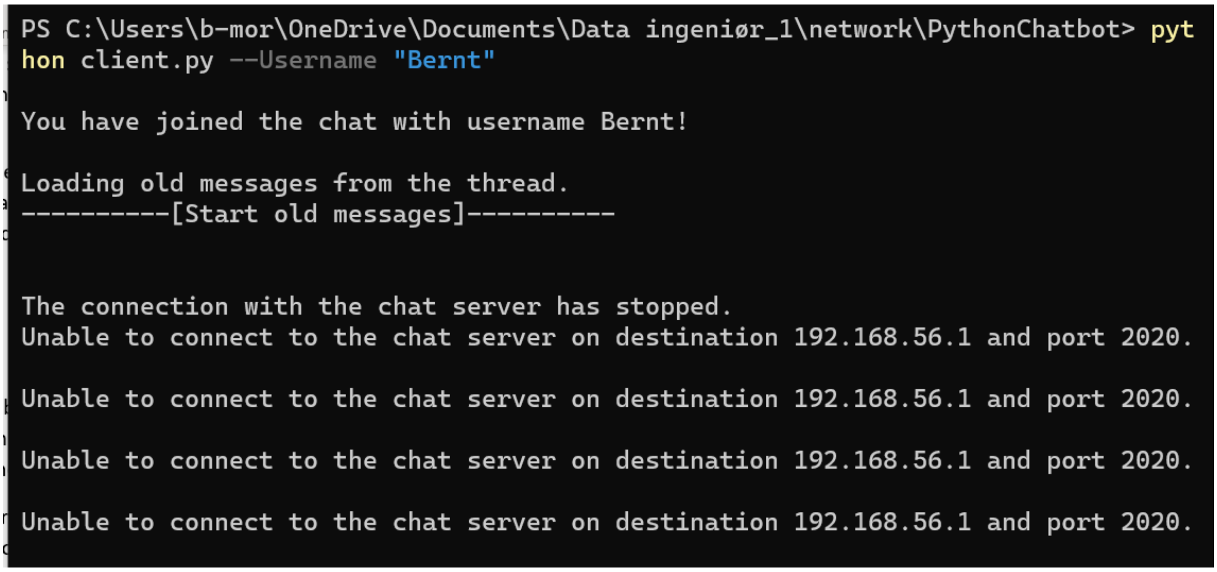
The client application also needs internet access in order to make requests to the Met API.

The requirements of client.py is only regarding the weatherbot which the client.py is starting. The rest of the bots are all independent and can be started without any other files or internet access. I have added functionality which replaces the weatherbot if client.py is started without some of the requirements.

All Python code created is tested with and developed for the python 3 interpreter. Please execute the programs with python 3.

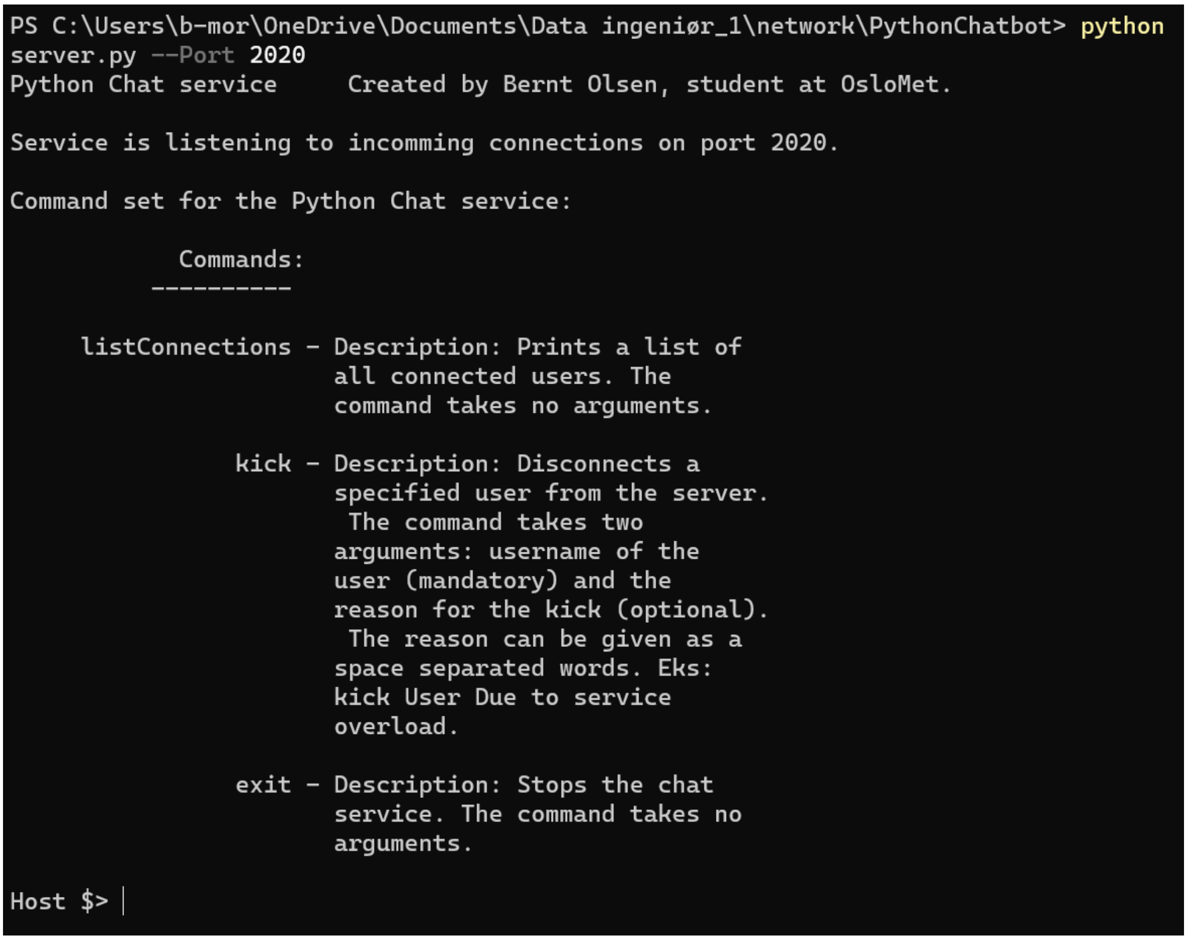
Start the solution

The first step is to start the server. If the server is not running, any instance of the client program will fail with the error message showed in figure 1. The server.py program takes one optional argument for the port on which it should listen for connections. The default value is 2020 and the argument should be specified as an integer. It must be a number between 0 and 65535. The address that this service is responding on is always set to all addresses associated with any of the network interfaces. Figure 2 shows the start screen of the server application after starting the program with the command “python server.py –Port 2020” in the directory with the server.py file.



```
PS C:\Users\b-mor\OneDrive\Documents\Data ingeniør_1\network\PythonChatbot> pyt
hon client.py --Username "Bernt"

You have joined the chat with username Bernt!

Loading old messages from the thread.
----------[Start old messages]----------


The connection with the chat server has stopped.
Unable to connect to the chat server on destination 192.168.56.1 and port 2020.

Unable to connect to the chat server on destination 192.168.56.1 and port 2020.

Unable to connect to the chat server on destination 192.168.56.1 and port 2020.

Unable to connect to the chat server on destination 192.168.56.1 and port 2020.
```

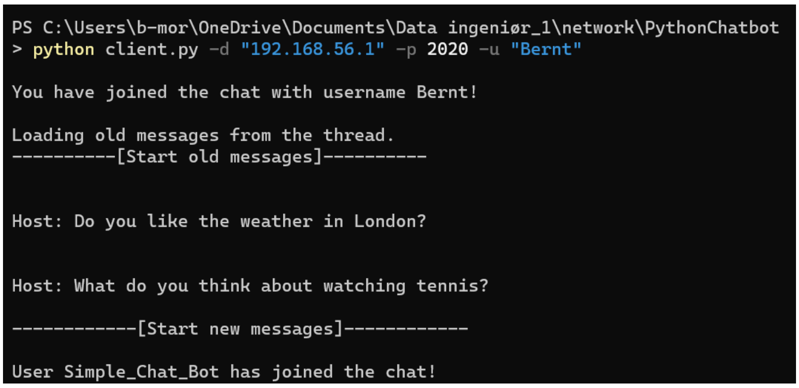
Figure 1- The client receives connection error messages



```
PS C:\Users\b-mor\OneDrive\Documents\Data ingeniør_1\network\PythonChatbot> python
server.py --Port 2020
Python Chat service     Created by Bernt Olsen, student at OsloMet.

Service is listening to incomming connections on port 2020.

Command set for the Python Chat service:

            Commands:
          ----------

     listConnections - Description: Prints a list of
                       all connected users. The
                       command takes no arguments.

                kick - Description: Disconnects a
                       specified user from the server.
                        The command takes two
                       arguments: username of the
                       user (mandatory) and the
                       reason for the kick (optional).
                        The reason can be given as a
                       space separated words. Eks:
                       kick User Due to service
                       overload.

                exit - Description: Stops the chat
                       service. The command takes no
                       arguments.

Host $> |
```

Figure 2- The server application has started

The next step is to start the client.py application. Please make sure that the client endpoint can reach the end system, which runs the server, with the given network connection. The client application also takes three optional arguments. The first is the destination argument (--Dest, -d) and takes an IPv4 address (String) to the server end system as argument. Default value is localhost. The second is the port argument and takes an integer representing the destination port that is used to connect the clients to the server. Default is port 2020. The third argument is the username (--Username, -u) that the user should be displayed with in the chat thread. The default value is “Chat_User”. Figure 3 shows the start display of the client program after it was started with the command “python client.py -d “192.168.56.1” -p 2020 -u “Bernt””.



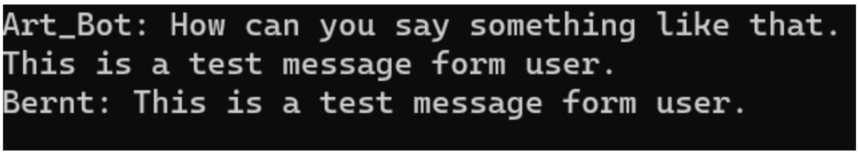
```
PS C:\Users\b-mor\OneDrive\Documents\Data ingeniør_1\network\PythonChatbot
> python client.py -d "192.168.56.1" -p 2020 -u "Bernt"

You have joined the chat with username Bernt!

Loading old messages from the thread.
----------[Start old messages]----------


Host: Do you like the weather in London?


Host: What do you think about watching tennis?

------------[Start new messages]------------

User Simple_Chat_Bot has joined the chat!
```

Figure 3- The client application has started

Both client.py and server.py can be started without any arguments if they are started on the same endpoint. The help text can be found by executing “python client.py --help” and “python server.py --help”.

Using the client application

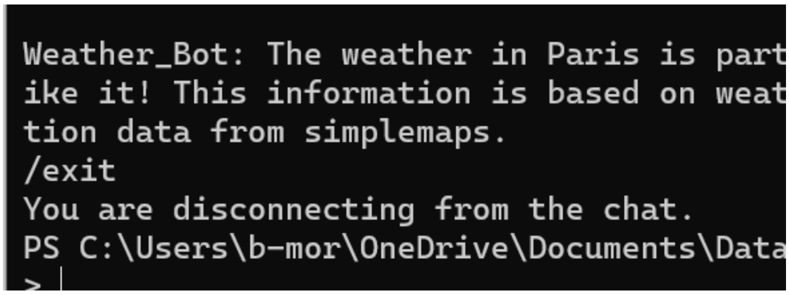
The client.py program starts chatbots and connect them to the chat server. This includes a user bot which takes input from the user running the program and sends it to the server. The user should be able to type a message and send it by pressing enter. The message is then printed out again below the line that the user wrote (see figure 4).



```
Art_Bot: How can you say something like that.
This is a test message form user.
Bernt: This is a test message form user.
```

Figure 4- The user has entered a message in the chat

The user can also close the program properly by entering the command “/exit” and pressing enter (see figure 5).

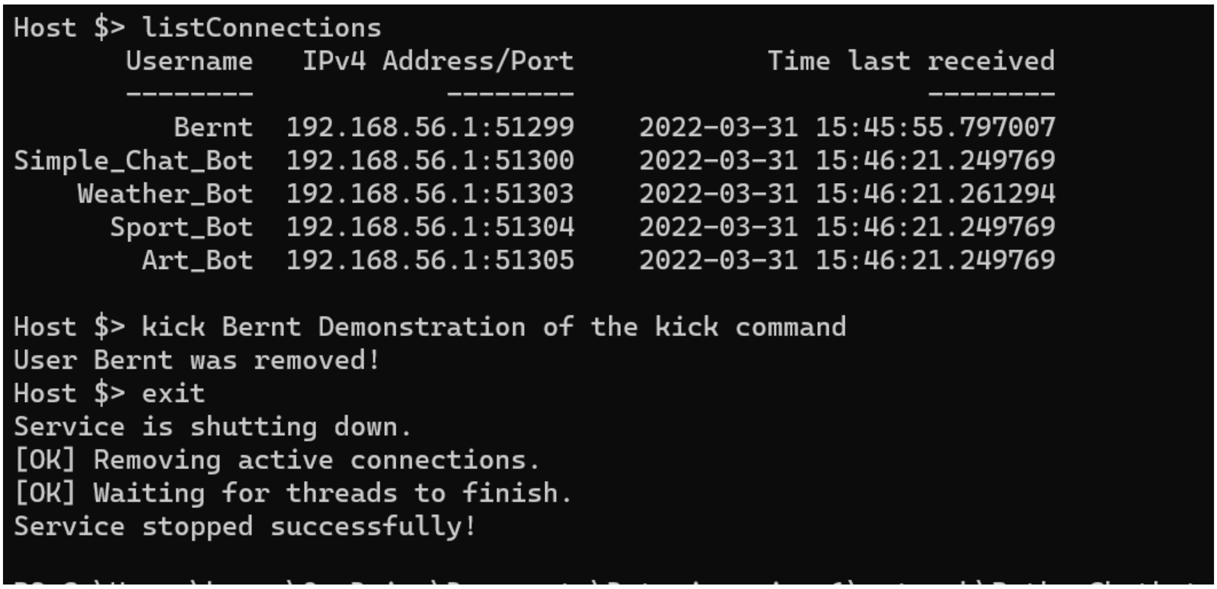


```
Weather_Bot: The weather in Paris is part
ike it! This information is based on weat
tion data from simplemaps.
/exit
You are disconnecting from the chat.
PS C:\Users\b-mor\OneDrive\Documents\Data
>
```

Figure 5- The exit command was entered by the user

Using the server application

There are three commands that can be used to maintain the single chat thread service. They can be entered into the terminal running the server. The first command is the “listConnections” command, which prints out a list of all users currently connected to the server. The second command is the “kick” command, which removes users from the chat. The command must be given the username of the user that should be kicked, as the first argument. Optionally, a reason can be provided after the username (see figure). The last command is the exit command. This command stops the chat service and ends all TCP connections. Figure 6 shows the commands in action.



```
Host $> listConnections
       Username   IPv4 Address/Port            Time last received
       --------            --------                      --------
          Bernt  192.168.56.1:51299    2022-03-31 15:45:55.797007
Simple_Chat_Bot  192.168.56.1:51300    2022-03-31 15:46:21.249769
    Weather_Bot  192.168.56.1:51303    2022-03-31 15:46:21.261294
      Sport_Bot  192.168.56.1:51304    2022-03-31 15:46:21.249769
        Art_Bot  192.168.56.1:51305    2022-03-31 15:46:21.249769

Host $> kick Bernt Demonstration of the kick command
User Bernt was removed!
Host $> exit
Service is shutting down.
[OK] Removing active connections.
[OK] Waiting for threads to finish.
Service stopped successfully!
```

Figure 6- The server console with executed commands

Chatbots

All chatbots created in this assignment, are started by the client.py program. There are totally 4 chat bots (SimpleChatBot, WeatherBot, SportBot, ArtBot) and one chat user which is a bot that takes user input (simulates a normal user). They are programmed as different classes where the bots inherit from the chat user class. The bots are in other words building upon the normal chat client, extending with methods used to generate responses automatically. Furthermore, the ArtBot, SportBot and the WeatherBot is inheriting from the ChatBot class which is the base for each bot added to the system. This systematic inheritance which is implemented here makes it simple to add new bots which have the basic bot functionality in addition to the special features added.

In order to create a response, each message is received by the bot socket is analysed by the classifyMsg method in the MsgAnalysis class. The methods execute some predefined tests which determine if the message contains certain subjects or activities, if it is a question (contains question words) or if it is a statement. The tests consist of sequences where the code is looping over the words in the message and comparing them with predefined words. There is also defined some Regular expression patterns used to parse the message in order to analyse the message. The message is tagged with a tag (enum type, see class Tags) for each positive test. The tags are used to classify the message and describes the content of the message.

There are also some words that are extracted from the message if they are matched. The MsgAnalysis class has the attribute`s location, activity and username. The location attribute is set if the location test is positive, and a location was identified (used for the weather bot). This could for example be a city name. The activity attribute is set if the message contains an activity and the username is set if the message is identified as a connect message (see section client below).

An example could be the message “What do you think about watching tennis?”. The first message that will be conducted is the question/statement test. This test determines if the message contains one of the question words defined in the regex pattern “questionWords”. Since “What” is in the pattern, this message will be tagged with the question tag, indicating that it is a question. The next test which will be positive is the is the subject test. This test will recognise the word “tennis” as an activity and a sport activity. Therefore, the activity and sport tags are added. In addition, the activity attribute of the MsgAnalysis object will be set to “tennis”. The last test that will be positive is the opinion test. This test checks for words which are indication that the message is asking for an opinion. In this case the words “do you think” is matched by the regex search and the opinion tag is added as a result.

The bots which instantiate this MsgAnalysis object for each received message, uses the tags set for the message to generate a response which is typical for the bot. The example message above is for example something the Sportbot and the Artbot is specialized to respond to. Those two bots have interesting responses for messages tagged with activity tags. The weather bot on the other hand responds with special responses to messages about weather. The process of generating the special response is executed by the getBotResponse method which is defined individually for each bot class.

We can turn back to the example with the tennis message above and observe how the getBotResponse method of the SportBot is handling the message. The generation of a response start by that the run method calls the generateResponse method (both methods are defined in the ChatBot class) which are common for all bots. The generateResponse method creates the MsgAnalysis object of the message and makes some common checks of the message, to determine if it is too complicated, it is a message which should not be responded to or is join message (a greeting message is sent if it is a join message).

The generateResponse method then calls the getBotResponse which starts by checking if the message is tagged as a question or a statement. The example message is a question, and the program continues by checking if it is a request for an opinion. It is a request for an opinion and the program further finds out that the message is tagged with activity. From those three if tests, the program has reached a point where there are defined responses for that message. The correct response list is then found for the corresponding activity set for the message object. From the list of responses, one random picked message is returned from the method. The procedure described above is similar for all bots and messages that the bots received.

The following snippet shows some of the added tests in the MsgAnalysis class:

```
    # Regex pattern used to detect messages which are too complicated.
    # A message is for example too complicated if it contains several sentences.
    complicationDetection = re.compile(".*: .*[\.\?\!\:][A-Za-z0-9\s]+")
    # A regex pattern used to determine if the message is asking a question.
    questionWords = re.compile("([Ww]hat)|([Ww]here)|([Ww]hen)|([Ww]hy)|([Ww]hich) \
                        |([Ww]ho)|([Hh]ow)|([Ww]hose)|([cC]an you)|([cC]ould you)|([Ii]s it)|([dD]o
you)|([wW]ould you)")

    # Words about the weather mapped to their coresponding tags
    weatherSubjects = {"temperature" : Tags.temperature, "weather" : Tags.weather, "hot" :
Tags.temperature,
                "cold" : Tags.temperature, "sunny" : Tags.weather}

    # List of regex strings used to determine if the messages makes a sugestion for an activity
    requestActivities = ["[wW]e could", "[wW]e should", "[cC]an we", "[cC]ould we"]

    # Activity words maped to their coresponding tags
    activitiesOpinion = {"sport" : Tags.sport, "art" : Tags.art, "football" : Tags.sport, "tennis" :
Tags.sport,
                  "draw" : Tags.art, "drawing" : Tags.art, "paint" : Tags.art, "painting" : Tags.art}

    # List of regex strings used to determine if the message asks for an opinion.
    opinions = ["[Dd]o you like", "[Cc]an you rate", "[Pp]lease rate", "[Dd]o you think", "[Hh]ow would
you rate", "What do you think about"]

    # List of regex string used to detect locations
    locationWords = ["[iI]n", "[aA]t"]
    # The strings in locationWords are substituted in this regex string.
    # It finds the word refering to a location
    locationReg = "{} (.*)[\s\!\.\?\,]|{} (.*)$"

    # Regex pattern used to detect join messages
    regJoinCase = re.compile("User (.*) has joined the chat!")
```

Explanation of the client program

The client program is consisting of several threads created with the threading module. There is one main thread which starts all the clients (four bots and one ChatUser) where each client is running in one thread. The ChatUser thread is also creating one more thread used to handle user input in parallel with the normal send/receive procedure. Each of the client threads are acting as one client and connects to the server as one individual user, simulating several users in a single-threaded chat.

Since each of the threads acts as one individual client, they each have their own client socket. All sockets are defined to use the IPv4 address family (socket.AF_INET) and the TCP protocol (socket.SOCK_STREAM). Furthermore, the four chatbots implements the exact same type of socket and executes the same code for sending, receiving and connecting. The run method for the bots is therefore defined in the ChatBot class and inherited by the other bots.

The sockets used by the bots are set to be blocking. This means that the calls to the socket API are waiting until the call is finished. For example, when the bot is sending a message and calls the clientSocket.send() method, it will wait until the data has been sent and the buffer has free space. It then adds as much of the message into the send buffer, as possible (MacMillan, n.d)(Kapoor, 2015). In this case I have programmed the send method of the bots such that they continue to call the send function until all messages in the send queues are sent. The same principle of blocking is appearing with the connect and receive calls (MacMillan, n.d)(Kapoor, 2015).

The ChatUser thread on the other hand uses a non-blocking client socket. This means that the connect, receive and send calls do not wait for the buffer to be emptied or filled. The send call adds as much of the message as possible to the send buffer and returns the number of bytes sent. If the buffer is full, then no data is sent, and the send method returns. The same appears for the receive call and the receive method. The receive call gets a fixed part of the data in the receive buffer (4096 bytes in this case) and returns without waiting. If the receive buffer does not contain any data, no data is fetched. Since it is very inefficient to call the send and receive functions without achieving anything, the select call is implemented in the main loop of the run method defined for the ChatUser (Hellmann, n.d).

The select function is used to determine if the send buffer has free space, if the receive buffer contains data or if the TCP connections has an error. It is in this case blocking for maximum 5 seconds, until any of the send or receive calls would have an effect or there is an error with the socket. A call to send and receive is then guarantied to send or fetch some data. Because the send buffer most likely will have free space while there is no data to send, the socket is only added for write checking by the select command if there are some messages to be sent from the client (Hellmann, n.d) (Kapoor, 2015). The result of this is that there could be a longer delay from the time the user entered a message until it is sent to the server since the send/receive thread might be blocking on the select command.

All clients are stopped when the user writes the command “/exit” in the chat. This action sets a thread-safe flag (threading.Event) to True, which in turn ends the while loop in the run method of the ChatUser thread. The client socket is then closed with the close() call and the thread is ended. The main thread resumes after the thread ends and continues by calling the initiateClosure on all the bot threads. This method also sets a thread-safe flag to true in order to stop the bot clients (Anderson, n.d).

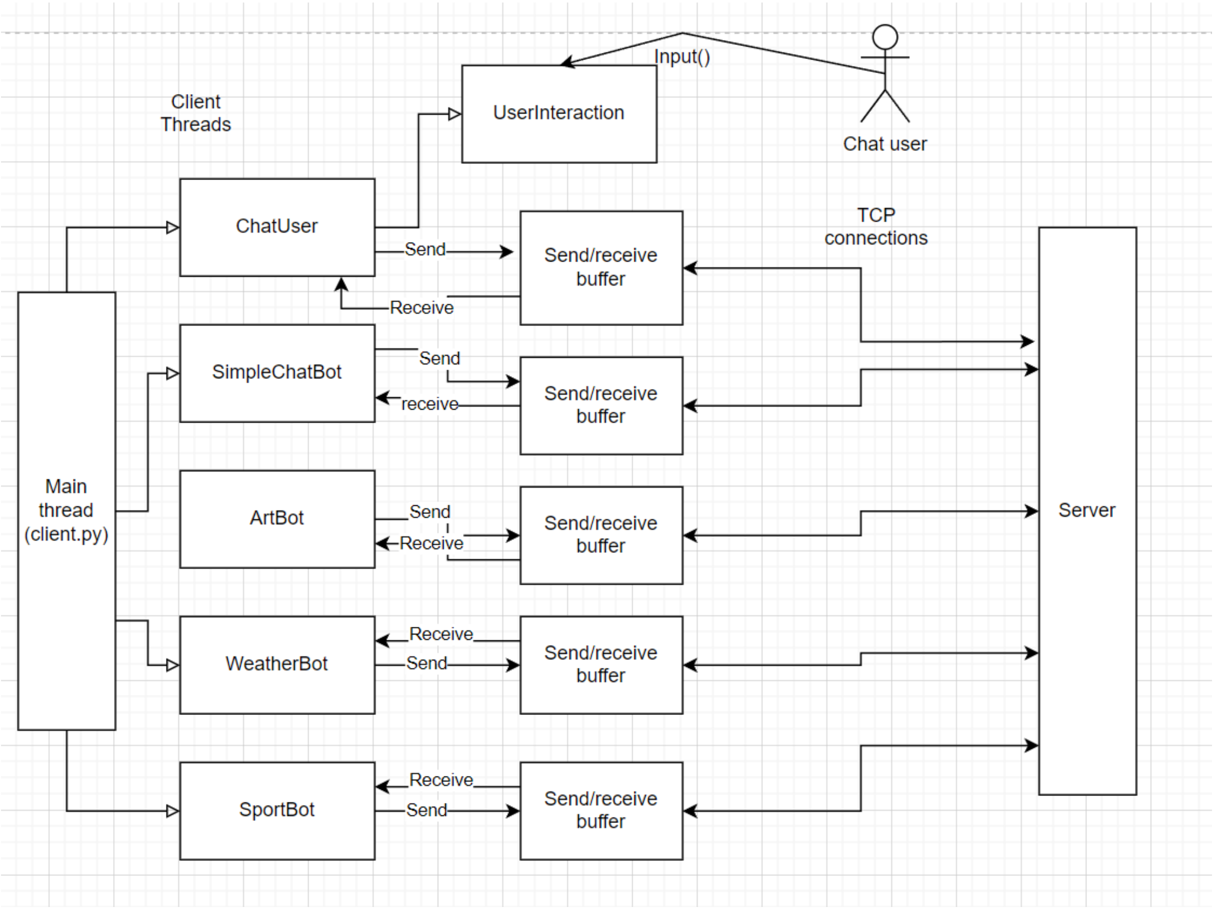


Figure 7- An overview of the client threads

Explanation of the server program

The server program server.py consists of several threads like client.py. In this case the initial thread is the user interaction thread handling input from the user. When the program starts, this thread starts the main thread which is the thread controlling the server socket and all connections with the server. The server socket is created with IPv4 as address family and TCP protocol. Furthermore, it is set to listen on the given port and on any address for the end system. The following code defines the socket (from the startServer method in server.py):

```
    # Defines the main server socket with the IPv4 address family (AF_INET)
     # and the TCP protocol (SOCK_STREAM) as domain and type
     self.serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
     # Binds the server socket to the given port and any address associated to any network card ont
he ensystem
     # running this program.
     self.serverSocket.bind(('', self.port))
     # The server starts listening on the given port.
     # The number of unaccepted connections to the server before the server refuses any new
connections, is 5.
     self.serverSocket.listen(5)
     # The server should be non-blokking
     self.serverSocket.setblocking(0)
```

Note that the address is set to ‘’, which indicates any address associated with the end system. The socket is set to be non-blocking like the client socket for the UserChat. This means that the send, receive and accept calls are not waiting and potentially raises the BlockingIOError exception if it should have been blocking (Kapoor, 2015). In this case the select function has been used again in order to determine if any of the connected sockets have data in the receive buffer, if they have non-empty send buffer or an error. The server socket is also added in the list of sockets that the select command should check the read buffer for. If the select command returns the server socket, then there is a new connection that should be accepted by the server.

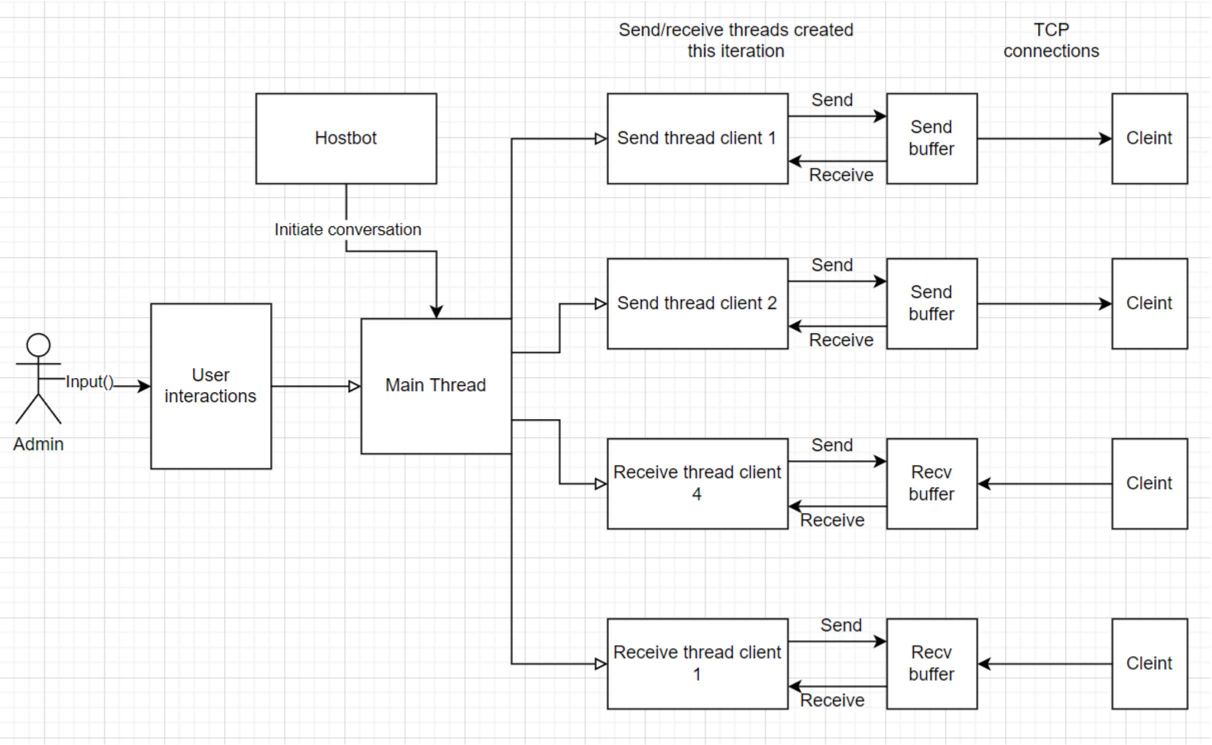


Figure 8 - An overview of the server threads

The main thread also starts a single thread for the host bot. The thread is simulating a user that is sending messages in the chat to initiate the conversation with the chat bots. A random new message will in this case be added in the chat thread every 30 seconds.

The main loop in the main thread is first executing the select command on the server socket which is the only socket in the ckeckReadable list (no sockets in writable or error lists). It is blocking until there is a new connection or the timeout (10 seconds) has passed, and the loop finishes the iteration. When the new connection is discovered, the select command returns the server socket and an accept thread is started. This thread executes the acceptConnection method which accepts the connection and adds the client socket to the list of active users.

In the next iteration the select command will check the send and read buffer (and if there are errors) of this new client. It returns the client socket in the readable list if it has data in the receive buffer, in the writable list if there is data in the send queue for the client and the send buffer is not full or in the error list if there is an error with the connection. If it is in the error list, the connection is removed. If it is in the readable list, a new receive thread is started for this socket in order to obtain data from the receive buffer of the socket. The receive thread is ended when one receive call has been made. If it is in the writable list, then a send thread is started and all data in the send queue associated with the socket will be added to the send buffer unless the buffer becomes full in the process. The send process/thread is ended when the send buffer becomes full. The select command will then detect when the buffer is empty again and initiate the send process again for that client. Figure 8 shows an overview of the threads created in the server program.

When new connections are accepted, each new client socket is associated to an instance of the ChatSocket class. This class contains some attributes which describes the client. Some of the attributes are username, last receive time stamp and the send queue. The send queue is important in this case since this is a data structure that contains all the messages that should be sent to the client. If there is messages in the send queue, then the socket is added to the checkWritable list in order to get the select function to check it`s send buffer. The queue gets populated with messages from the receive threads which forwards the received messages to all send queues of the connected users. Since the send threads also access the send queue, a race condition is created. This is not a problem in this case since the data structure used is thread-safe (Queue object from the queue module) (Anderson, n.d).

The server is closed by entering the “exit” command in the terminal. This causes the user interaction thread to call the stopService which first removes the server socket from the readable list to prevent new connections. Second all TCP connections are ended before the threads are closed. The clients are all notified that the chat service has stopped.

Another important feature of the chat service is that the chat thread is stored in the memory of the server (the history variable). If a user logs off the chat and joins again, all previous messages in the thread are sent to the client again. The history variable is reset when the server is restarted. An improvement would be to save the chat data to the disk on the server.

The following code snippet displays the first part of the main loop in the mainThread method (server.py file):

```
    while not self.stopApplication.is_set() and (len(self.checkWritable) > 0 or len(self.checkReadable)
> 0):
        # Continue handling connections while the stopApplication falg is not set and the there are
sockets
        # that should be checked for readablity or sending
        try:
            # Check if the any socets have data in their inbound buffer, if they have free space in thye
            # outbound buffer or if there is any error associated with the socket.
            # The timeout argument is set to 10 seconds. This means that this function will block for 10
            # seconds if no messages are received or sent
            readable, writable, err = select.select(self.checkReadable, self.checkWritable,
self.checkError, 10)
        except OSError as E:
            # If the select method raises an OSError, the application is stopped.
            logging.error(f"The select function raised the following exception: {E}")
            readable, writable, err = []
            print(f"Fatal error in main thread. Program is closing: {E}")
            # Set both flags to end the loop in main thread and in the user interaction thread
            self.stopApplication.set()
            self.stopUserInteraction.set()
```

The simple application layer protocol

In order to send messages between the server and the clients, some rules in the application layers must be followed. The first is that the first message from the client to the server must be a connect message containing only the username. This message is identified by the server and the username is added to its corresponding ChatSocket. If the first message is not the connect message, the connection will be closed again.

Another rule is that each message must end with the end of message code (END_OF_MSG) to indicate the end of a message. The code is “::EOMsg::”. This way several the client and the server can identify separate messages and save the rest of a message if the end was not received in the same receive cycle.

The code snippet below shows the part of the receive method in server.py where the received data is decoded and splitted with the end of message code as delimiter:

```
# Extracts the rest of the previous reception. This variable contains
# a non empty string whenever the previous reception received a portion of the next message
# in the same process.
data_recv = curChatUser.recvRest
# Definition of a variable for the new data
cur_recv = ""

try:
    # Read from the buffer of the client socket
    cur_recv = cliSock.recv(4096).decode()
except OSError as E:
    # If an OS exception is raised, log the error and endd the connection
    self.connectionErrorHandling(curChatUser, cliSock, str(E))
    return

if len(cur_recv) == 0:
    # If the recv method returned nothing, then the connection is closed.
    # The data that was sent before an EOMsg was found will be dropped
    self.connectionErrorHandling(curChatUser, cliSock)
    return

# The received data is added to the data from the previous receive procedure
data_recv = data_recv + cur_recv

logging.info(f"Data received: {data_recv}")

# Cleaning up the received data and create a list of all messages
# contained in the received bytes.
msgList = data_recv.replace("\n", "").split(self.END_OF_MSG)
```

Results

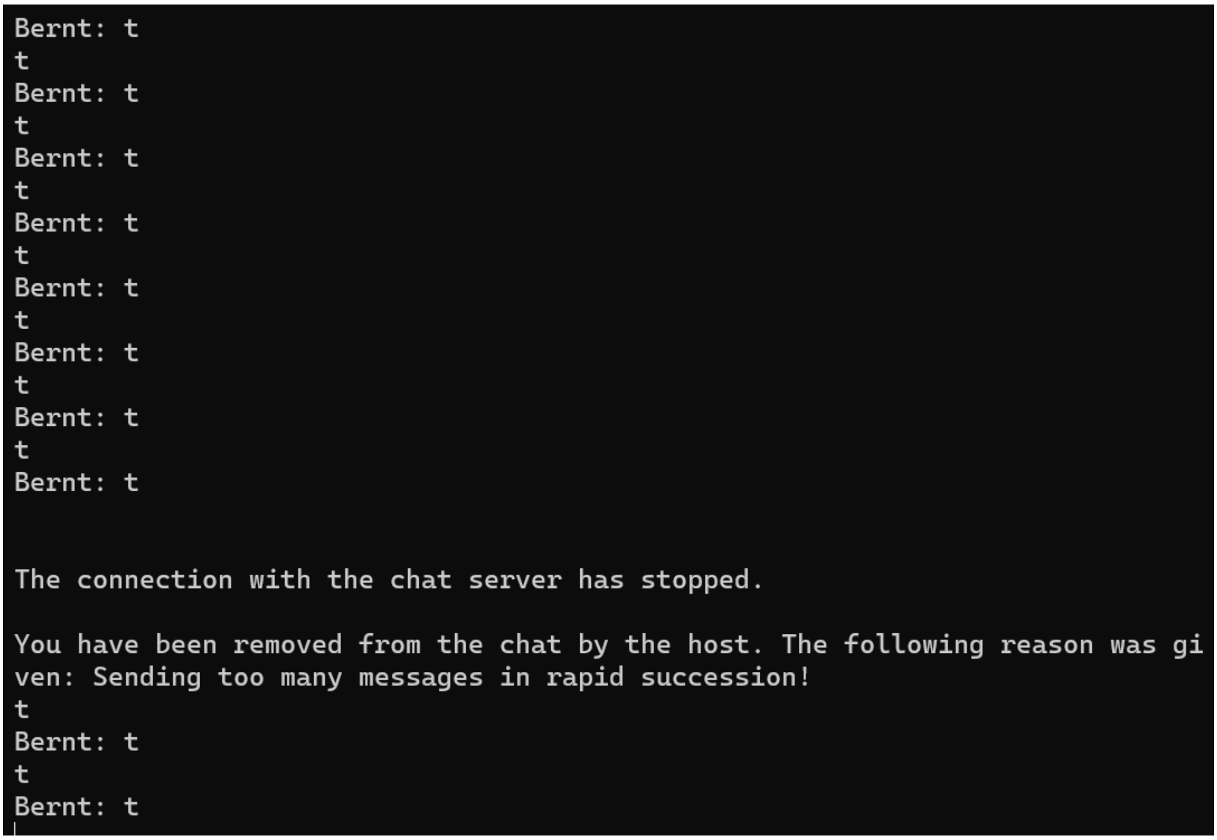
I tested this solution by running both server and client programs on my local computer. First I started the programs and observed the conversations that were initiated in the chat. The text below is the chat thread displayed on the client after a short while:

```
PS C:\Users\b-mor\OneDrive\Documents\Data ingeniør_1\network\PythonChatbot>python client.py
--Port 2020 --Dest "192.168.56.1" --Username "Bernt"

You have joined the chat with username Bernt!

Loading old messages from the thread.
----------[Start old messages]----------


Host: Do you like the weather in London?

------------[Start new messages]------------

User Simple_Chat_Bot has joined the chat!

Simple_Chat_Bot: Hello Bernt

User Weather_Bot has joined the chat!

User Sport_Bot has joined the chat!

Weather_Bot: Hello Bernt.

Sport_Bot: A good day to you, Bernt!

User Art_Bot has joined the chat!

Art_Bot: How are you today Bernt


Host: What do you think about watching tennis?

Simple_Chat_Bot: I am not sure, try to ask someone else.

Weather_Bot: Let me make up my mind first!

Sport_Bot: I love all kind of sport, and I am realy paying a lot of attention to tennis at the moment.

Art_Bot: I am not sure yet. I need to try it first.
```

```
Host: Do you like to play tennis?

Simple_Chat_Bot: I think it is nice!

Weather_Bot: I dont`t like it.

Sport_Bot: I could play tennis all day.

Art_Bot: It looks like a cool activity. I would like to try it once.


Host: It is cloudy in Oslo!

Simple_Chat_Bot: I did not know that.

Weather_Bot: How can you say something like that.

Sport_Bot: How can you say something like that.

Art_Bot: How can you say something like that.


Host: Is it sunny in Oslo?

Simple_Chat_Bot: I do not want to talk about the weather. It is boring and always depressing!

Sport_Bot: There is nothing called bad weather!

Art_Bot: I can`t complain. It is nice enough for me!

Weather_Bot: The temperature in Oslo is 1.8 degree celcius! The sky is wery cloudy. This information
is based on weather data from MET Norway and location data from simplemaps.


Host: Can we paint a painting?

Simple_Chat_Bot: Are you sure about that?

Weather_Bot: Are you sure about that?

Sport_Bot: Please, talk about something else! Maybey we can watch sport?

Art_Bot: That is a good idea.
```

From the result displayed above, we can first see the old messages in the thread which were sent in the chat before the user connected to server. After the old messages the connect messages for the bots are displayed among their greetings to the connected user (bots do not send greetings to each other). Furthermore, some conversations can be observer. Some of the conversations seem to be successful. An example is the conversation started with “Is it sunny in Oslo?”. It looks like all bots create a good response for the message and the weather bot gets the weather data from the Met API. But on the other hand, there are conversations where three bots send the same message (see the “It is cloudy in Olso message”). This is due to that there is little handling of statement messages in the bots.

Another feature of the solution is that users get automatically kicked if they send too many messages in short succession (spamming). I tested this by entering the letter “t” in the chat in short succession. The result can be seen in figure 9.



```
Bernt: t
t
Bernt: t
t
Bernt: t
t
Bernt: t
t
Bernt: t
t
Bernt: t
t
Bernt: t
t
Bernt: t


The connection with the chat server has stopped.

You have been removed from the chat by the host. The following reason was gi
ven: Sending too many messages in rapid succession!
t
Bernt: t
t
Bernt: t
```

Figure 9 - The user is kicked for spamming

Conclusion

The results show that the solution is working on my computer. I have also tested that it works on an ubuntu docker container. However, during the work with the assignment, I realized that there is much more room for improvement. I would for example try to implement a more sophisticated application layer-protocol for the server-client connections. Another improvement would be to add the bots to the server program so the administrator can start and stop the bots. The way the bots are added to the client application now is not optimal since they block on receive when the stop signal is sent. This can cause a long delay until the host sends the next message which eventually terminates the bot.

References

McMillan, G. (n.d). Socket Programming HOWTO. Python Software Foundation. https://docs.python.org/3/howto/sockets.html [31.03.2022]

Kapoor, V. (2015, May). Understanding Non Blocking I/O with Python — Part 1. Medium. https://medium.com/vaidikkapoor/understanding-non-blocking-i-o-with-python-part-1-ec31a2e2db9b [31.03.2022]

Hellmann, D. (n.d). select – Wait for I/O Efficiently. PyMOTW. http://pymotw.com/2/select/ [31.03.2022]

Anderson, J. (n.d). An Intro to Threading in Python. RealPython. https://realpython.com/intro-to-python-threading/ [31.03.2022]