



**Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Curso de Engenharia de Software**

**Projeto de pesquisa - Criando ambientes virtuais de
conversação com uso de sockets**

**Fundamento de Redes de Computadores
Professor: Fernando William Cruz**

**Autores:
Eduardo Afonso Dutra Silva - 190012307
Rafael Cleydson da Silva Ramos - 190019085
Thiago Sampaio de Paiva - 190020377**

Brasília, DF



SUMÁRIO

Introdução	3
Metodologia utilizada	3
Descrição da solução	4
Descrição do servidor	5
Descrição do cliente	13
Conclusão	16
Experiência da equipe	17
Referências	18

1. Introdução

O presente relatório tem o objetivo de fomentar o estudo prático dos conceitos ensinados na disciplina de Fundamentos de Redes de Computador do curso de engenharia de software da Universidade de Brasília, tendo como escopo a uso de sockets que utilizem o protocolo TCP IP para a criação de salas de bate-papo virtuais em tempo real.

A solução que será tratada neste relatório foi implementada utilizando a linguagem de programação Python com a system call select para fazer a comunicação *full-duplex* entre servidor e cliente.

Dentre as funcionalidades desenvolvidas estão a criação, entrada e saída de uma sala de bate-papo, além das listagens de canais disponíveis e de usuários conectados na presente sala.

2. Metodologia utilizada

A metodologia seguida para realizar o desenvolvimento do projeto foi basicamente realizar encontros síncronos com todos os membros da equipe para que as decisões, implementações e mudanças fossem propostas e debatidas com a ciência e aval de todos os indivíduos.

O primeiro encontro foi realizado no dia 30/04/2022 e serviu basicamente para trocar experiências sobre os conteúdos abordados no projeto, definir qual linguagem de programação seria utilizada e então começar a implementação. Com tudo definido, um membro da equipe compartilhou o código-fonte com a extensão *Live Share* para o editor de texto *Visual Studio Code*, assim os demais membros puderam navegar, ler e editar os arquivos, sendo essa a metodologia adotada durante todo o processo de desenvolvimento do projeto.

Ao final do primeiro encontro havia sido desenvolvida uma comunicação full-duplex simples feita por terminal, onde vários usuários poderiam se conectar em uma mesma sala. Ficou definido que a partir deste ponto cada membro da equipe faria uma pesquisa sobre conexões com *websocket* para que houvesse a comunicação com uma interface web.

O segundo encontro ocorreu no dia 02/05/2022. Todos os membros da equipe se reuniram novamente e informaram os resultados das pesquisas realizadas.

Nenhum dos membros conseguiu realizar a conexão de um websocket com o socket desenvolvido pela equipe, por isso a ideia de fazer uma interface web foi abandonada.

O foco então passou a ser refatorar o código já feito e desenvolver novas funcionalidades para o bate-papo, tendo como objetivo fazer toda a interação do usuário ser feita através do terminal. Foram desenvolvidas as funcionalidades de entrar e sair de um canal, listar os usuários conectados no canal atual e também a funcionalidade de listar todos os comandos juntamente com a descrição de cada um.

A última reunião ocorreu no dia 03/05/2022 e o foco foi finalizar o projeto. Foi adicionada a funcionalidade de criar um novo canal, listar todos os canais criados até o momento. Este encontro também foi marcado por refatorações de código e a correção de erros e exceções. Além disso, foi elaborado o presente relatório e a apresentação de slides.

3. Descrição da solução

Para a implementação da solução, tanto do cliente, como do servidor, a linguagem escolhida foi o Python juntamente com a biblioteca Click, especificada no arquivo *requirements.txt*, que fornece um ferramentas para construção de um programa de Interface por Linha de Comando CLI. Desta forma, a área de ajuda que exemplifica como rodar a solução é mostrada na Figura 1 e Figura 2. O código da solução está disponível [aqui](#).

```
→ final-redes git:(main) python run_server.py --help
Usage: run_server.py [OPTIONS]

Options:
  -i, --ip TEXT      Ip address of the server
  -p, --port INTEGER Port of the server
  -m, --max INTEGER  Max connections of the server
  --help             Show this message and exit.
```

Figura 1: Ajuda para executar o servidor.

```
→ final-redes git:(main) python run_client.py --help
Usage: run_client.py [OPTIONS]

Options:
  -i, --ip TEXT      Ip address of the server
  -p, --port INTEGER Port of the server
  --help             Show this message and exit.
```

Figura 2: Ajuda para executar o cliente.

Como mostrado nas figuras, para executar tanto o cliente, como o servidor, é possível informar a opção do IP do servidor (--ip), porta do servidor (--port), o número máximo de conexão para o servidor (--max) - Essas opções são opcionais, se não informar alguma opção o valor *default* é atribuído, é mostrado na Figura 4. Sendo assim, a Figura 3 mostra um exemplo da solução executando. Além disso, também é possível utilizar o Telnet como cliente.

```
* final-redes git:(main) python run_server.py -i 127.0.0.1 -p 7777 -m 10
Aguardando em ('127.0.0.1', 7777)
[<socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 7777)>]:
('Nova conexão',)
[]

* final-redes git:(main) python run_client.py -i 127.0.0.1 -p 7777
Conexao concluida. Digite --HELP para ajuda.
--HELP
-----
--CREATE <NomeCanal> <CapacidadeDeUsuarios> <NomeUsuario> - Create Channel
--JOIN <NomeCanal> <NomeUsuario> - Join Channel
--OUT - Exit Channel
--CHANNELS - List available channels
--LIST - List user in channel
--HELP - Show all commands description
-----
```

Figura 3: Solução executando

```
import click
from server.server import Server

@click.command()
@click.option('--ip', '-i', default='127.0.0.1', help='Ip address of the server')
@click.option('--port', '-p', default=7777, help='Port of the server')
@click.option('--max', '-m', default=10, help='Max connections of the server')
def main(ip, port, max):
    try:
        server = Server(ip, port, max)
        server.start()
    except KeyboardInterrupt:
        server.close()

if __name__ == '__main__':
    main()
```

Figura 4: Arquivo responsável por rodar o servidor

3.1. Descrição do servidor

Assim, o servidor é implementado na classe `Server` presente na pasta `server`. Nas figuras 5 e 6 podemos notar alguns aspectos chaves para o funcionamento da sala de bate papo, a Figura 5 representa a classe implementada que representa um servidor TCP/IP.

```

class Server:
    server: socket.socket
    ip: str
    port: int
    max_connections: int
    inputs: list[socket.socket]
    outputs: list[socket.socket]
    clients: list[socket.socket]
    channels: Channels

    def __init__(self, ip, port, max_connections):
        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.setblocking(0)
        self.ip = ip
        self.port = port
        self.max_connections = max_connections
        self.inputs = [ self.server ]
        self.outputs = []
        self.clients = []
        self.channels = Channels(self.server, self.outputs)

    def start(self):
        self.server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.server.bind((self.ip, self.port))
        self.server.listen(self.max_connections)
        self._run()
        self.close()

    def close(self):
        print("Fechando conexões")
        for client in self.clients:
            client.close()
        self.server.close()

```

Figura 5: Configurações básicas do socket do servidor

A classe representada na Figura 5 é responsável por toda a configuração necessária para o pleno funcionamento da socket do servidor. A classe `server` em sua inicialização cria um socket de protocolo TCP/IP na linha 12 por meio do comando `socket.socket(socket.AF_INET, socket.SOCK_STREAM)`, e logo após é definido que não haverá sockets bloqueantes entre elas. São recebidos os parâmetros `ip`, `port` e `max_connections`, tais parâmetros são fundamentais para a execução do `bind` e do `listen` do socket. O `bind` é feito no `ip` e porta especificados, sendo o padrão `'localhost'` e `7777`. Já o `listen` é feito através do `max_connections`, esse comando diz ao socket o máximo de requisições de conexão que queremos enfileirar.

Nota-se também outros atributos importantes para o funcionamento do servidor, como inputs, outputs, clients e channels, que são, respectivamente, as sockets a serem lidas, sockets que vão receber valores do server, lista de sockets clientes que estão conectadas no servidor, channels é a classe onde fica toda a lógica de comunicação entre servidor cliente.

A screenshot of a code editor with a dark background and light-colored text. The code defines a 'Server' class with attributes for server socket, IP, port, max connections, and lists for inputs, outputs, clients, and channels. It also includes a '_run' method that uses 'select.select' to manage multiple connections non-blockingly.

```
class Server:
    server: socket.socket
    ip: str
    port: int
    max_connections: int
    inputs: list[socket.socket]
    outputs: list[socket.socket]
    clients: list[socket.socket]
    channels: Channels

    ...

    def _run(self):
        print("Aguardando em ", (self.ip, self.port))

        while self.inputs:
            readable, writable, exceptional = select.select(self.inputs, self.outputs, [])

            for r in readable:
                self._handle_input(r)

            for w in writable:
                self.channels.send_channel_message(w)

            for e in exceptional:
                self._handle_exceptions(e)
```

Figura 6: Configurações básicas do socket do servidor

Na figura 6, pode-se notar a função `_run` pertencente ao server, que realiza o gerenciamento de mais de uma conexão por vez através da *syscall select*. Esse gerenciamento é possível devido ao socket TCP IP não bloqueante criado anteriormente. Os argumentos enviados ao select são três listas contendo os canais de comunicação para o monitor, a primeira lista é referente ao recebimento de dados dos objetos (*sockets*) enviados, a segunda é referente aos objetos (*sockets*) que vão receber mensagem do servidor, e a terceira recebe qualquer exceção que haja na comunicação. Foram feitos alguns métodos para lidar com esse retorno do select, sendo eles: o `_handle_input`, `channels.send_channel_message` e o `_handle_exceptions`.

O método `_handle_input` recebe como parâmetro um socket e tem a responsabilidade de lidar com a entrada de dados, seja uma nova conexão ou uma mensagem submetida por um cliente. Assim, o tratamento da entrada de dados do servidor é mostrado na Figura 7.

A screenshot of a code editor with a dark background and light-colored text. The code defines a `Server` class with various attributes and methods. The `_handle_input` method is the central logic, branching into `_manage_connection` for new connections and `_manage_message` for incoming data. The `_manage_message` method includes logic to decode data, check for empty messages, and handle specific commands like "GET".

```
class Server:
    server: socket.socket
    ip: str
    port: int
    max_connections: int
    inputs: list[socket.socket]
    outputs: list[socket.socket]
    clients: list[socket.socket]
    channels: Channels

    ...

    def _handle_input(self, input):
        if input is self.server:
            self._manage_connection(input)
        else:
            self._manage_message(input)

    def _manage_connection(self, s):
        connection, _ = s.accept()
        connection.setblocking(0)

        self.inputs.append(connection)
        self.clients.append(connection)

        connection.sendall(b"Conexao concluida. Digite --HELP para ajuda.\n")
        logger(s, "Nova conexão")

    def _manage_message(self, conn):
        data = conn.recv(1024)

        try:
            if data:
                message = data.decode('utf-8')
                if len(message.strip()) == 0:
                    return

                command = message.split()[0]
                if command == "GET":
                    logger(conn, "Requisitando pagina", data)
                    self._handle_http_get(conn, data)
                else:
                    self.channels.manage_commands(conn, command, data)
            return
        except Exception:
            logger(conn, "Desconectando...")
            self._close_connection(conn)
```

Figura 7: Tratamento da entrada de dados no servidor.

Como é possível observar, a entrada de dados pode ser de uma nova conexão, tratada no método `_manage_connection`, ou de uma mensagem do cliente, tratada em `_manage_message`.

O `_manage_connection` recebe um socket (s) como parâmetro e aceita uma conexão, essa nova conexão é salva na variável `connection` é salva na lista de inputs para o select e na lista `clients` e, por fim, é enviado uma mensagem de conexão estabelecida com sucesso para o cliente.

Já em `_manage_mensagem` é tratada a mensagem do cliente. A leitura acontece no método `recv` do socket do cliente, lendo até 1024 bytes de dados. Após a leitura dos dados, acontece uma validação e, se caso uma exceção for lançada, o cliente é desconectado do servidor para garantir a sua integridade. O servidor aceita dois tipos de mensagem: um HTTP GET, que requisita um arquivo; ou uma mensagem que interage com os canais de bate-papos.

A interação com o HTTP GET é mostrado na Figura 8. Já a interação com o canal de bate-papo é exemplificada na Figura 9 adiante.



```
class Server:
    server: socket.socket
    ip: str
    port: int
    max_connections: int
    inputs: list[socket.socket]
    outputs: list[socket.socket]
    clients: list[socket.socket]
    channels: Channels

    ...

    def _handle_http_get(self, conn, msg):
        method, filename, *headers = msg.split()
        filename = "index.html" if filename == '/' else filename[1:]
        pages_path = pathlib.Path("./pages")
        file = pages_path / filename

        if file.exists():
            status = b'200 OK'
        else:
            status = b'404 Not Found'
            file = pages_path / 'notfound.html'

        conn.sendall(b'HTTP/1.1 ' + status + b'\nConnection: Closed\n\n')
        conn.sendall(file.read_bytes())

        self._close_connection(conn)
```

Figura 8: Tratamento do HTTP GET.

O [cabeçalho HTTP](#) fornece as informações suficientes para o método `_handle_http_get`, sendo utilizado o verbo GET e o caminho para o recurso. Para que o servidor funcione com o navegador, é necessário retornar uma [HTTP Response](#). A partir do caminho, é possível verificar se o arquivo existe na pasta

pages, se sim, o arquivo é retornado com *status code 200*, caso contrário, uma página HTML de não encontrado com *status code 404* é retornado.

Os recursos chaves para que o servidor funcione foram explicados. Finalmente, a lógica de negócio do bate-papo foi implementada no arquivo *server/channel.py* e será discutida a seguir.

O objeto *ChannelCommands*, exemplificado na Figura 9, representa todos os comandos disponíveis para o usuário, o formato que deve ser digitado no terminal e a descrição do ação que o comando realiza. Esta foi a forma encontrada para facilitar o entendimento dos objetivos do usuário por parte do servidor, assim é possível diferenciar qual funcionalidade o usuário deseja utilizar.

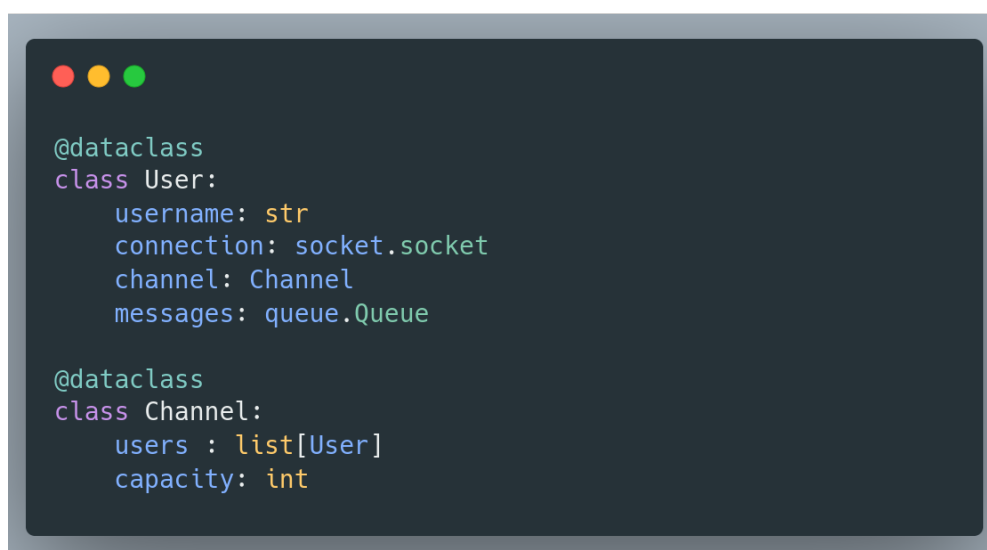


```
ChannelCommands = {
    "--CREATE <NomeCanal> <CapacidadeDeUsuarios> <NomeUsuario>": " - Create Channel",
    "--JOIN <NomeCanal> <NomeUsuario>": " - Join Channel",
    "--OUT": " - Exit Channel",
    "--CHANNELS": " - List available channels",
    "--LIST": " - List user in channel",
    "--HELP": " - Show all commands description"
}
```

Figura 9: Comandos disponíveis para o usuário

Como é possível observar, o usuário pode **criar**, **entrar** e **sair** de um canal de bate-papo, além de **listar os canais ativos**, **listar os usuários no canal** e **pedir por ajuda**.

Para fornecer essas funcionalidades, foi necessário criar duas estruturas que representam o usuário (User) e o canal (Channel), que são mostradas na Figura 10.



```
@dataclass
class User:
    username: str
    connection: socket.socket
    channel: Channel
    messages: queue.Queue

@dataclass
class Channel:
    users : list[User]
    capacity: int
```

Figura 10: Classes de dados de usuário e canal

As classes de dados Users e Channel referem-se a alguns dados importantes para lidar com as lógicas de comunicação implementadas. A classe User contém informações sobre o nome do usuário, o seu socket, o canal atual e a fila de mensagens que ainda não foram enviadas do usuário. Já a classe Channel é responsável por manter atualizada a lista de usuários presentes em determinado canal e armazenar a capacidade máxima de usuários em determinado canal.

A classe Channels é quem lida com toda a lógica e gerenciamento dos canais, desde sua criação, utilização e deleção. Além disso, também são gerenciados os usuários que podem estar ou não presentes em canais. A Figura 11 apresenta a classe Channel juntamente com a implementação da identificação e execução do comando enviado pelo usuário.

```
class Channels:
    server: socket.socket
    channels: dict[str, Channel]
    users: dict[socket.socket, User]
    outputs: list[socket.socket]

    def __init__(self, server, outputs):
        ...

    def manage_commands(self, conn: socket.socket, command: str, data: bytearray):
        message = data.decode("utf-8")
        command = command.upper()

        if command == "--CREATE":
            self.create_channel(conn, message)
            return
        elif command == "--JOIN":
            self.join_channel(conn, message)
            return
        elif command == "--HELP":
            self.explain_commands(conn)
            return
        elif command == "--CHANNELS":
            self.list_channels(conn)
            return

        if not self.is_user_in_channel(conn):
            self.send_server_message(conn, f"[ERRO] - E necessario entrar em uma sala para utilizar o comando {command}\n")
            return

        if command == "--LIST":
            self.list_users(conn)
        elif command == "--OUT":
            self.exit_channel(conn)
        else:
            self.enqueue_message(conn, message)
```

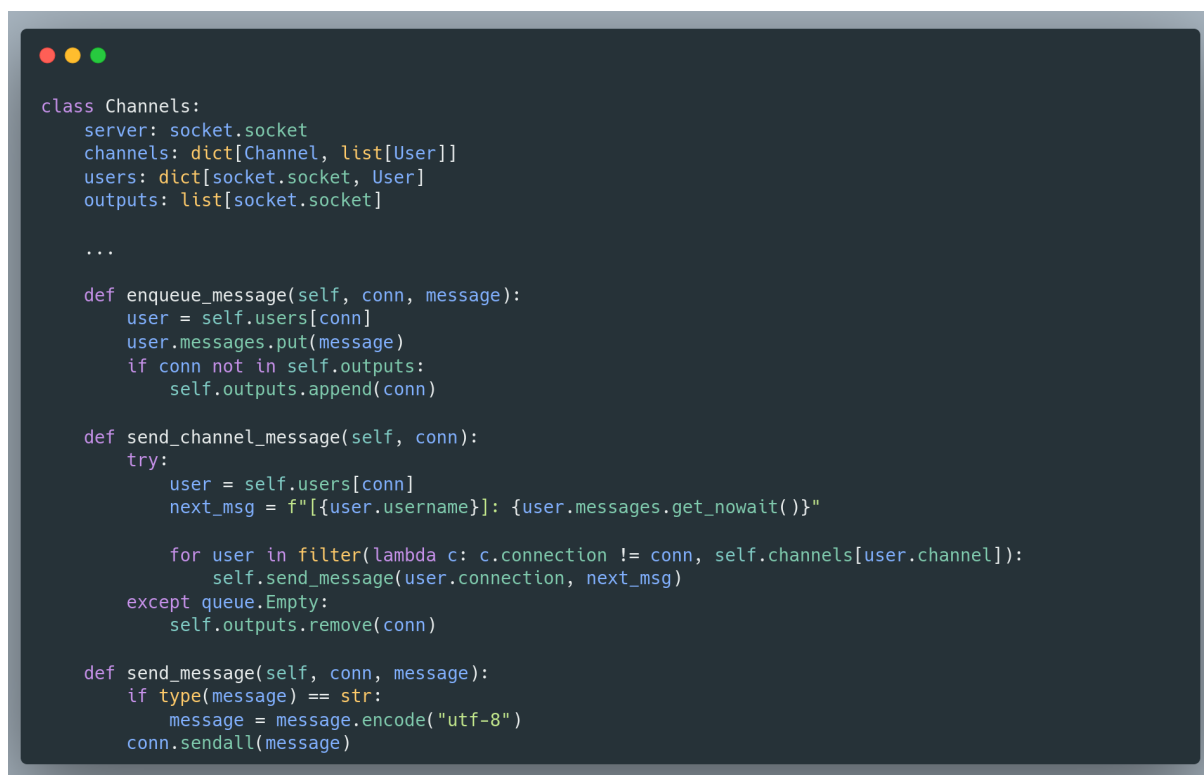
Figura 11: Classe Channel com o método que trata os comandos do usuário.

Na função *manage_commands*, os comandos digitados pelo usuário são identificados e direcionados para o método correto para cada caso, sendo que se o usuário não informar nenhum comando, o sistema interpreta que o que foi digitado

trata-se de uma mensagem que deve ser enviada ao canal em que o mesmo está conectado.

Todos os métodos chamados pelo mapeamento dos comandos do usuário apresentam um extenso tratamento de erros que podem ser cometidos pelo usuário, a fim de evitar qualquer situação que possa vir a cancelar a conexão indevidamente. Além de haver uma pequena validação no próprio *manage_commands* para que comandos que só devem ser utilizados dentro de canais não sejam executados fora de um.

Desta forma, os métodos necessários para o envio da mensagem do usuário para os usuários presentes no canal são mostrados na Figura 12.



```
class Channels:
    server: socket.socket
    channels: dict[Channel, list[User]]
    users: dict[socket.socket, User]
    outputs: list[socket.socket]

    ...

    def enqueue_message(self, conn, message):
        user = self.users[conn]
        user.messages.put(message)
        if conn not in self.outputs:
            self.outputs.append(conn)

    def send_channel_message(self, conn):
        try:
            user = self.users[conn]
            next_msg = f"[{user.username}]: {user.messages.get_nowait()}"

            for user in filter(lambda c: c.connection != conn, self.channels[user.channel]):
                self.send_message(user.connection, next_msg)
        except queue.Empty:
            self.outputs.remove(conn)

    def send_message(self, conn, message):
        if type(message) == str:
            message = message.encode("utf-8")
        conn.sendall(message)
```

Figura 12: Classe Channels com os métodos de envio de mensagem para o canal

Para realizar o envio de mensagem, primeiro é necessário enfileirar a mensagem na fila de mensagens do usuário e adicionar o socket na lista de *outputs* para ser selecionado no *select* do servidor, como é mostrado no método *enqueue_message*. Utilizar o *select* é necessário para garantir a conexão Full Duplex do servidor sem a necessidade de utilizar *threads*.

Assim, o *select* multiplexa os outputs e o método *send_channel_message* é acionado com a mensagem do usuário. A partir da conexão do socket é possível encontrar o usuário por meio do dicionário *users* e resgatar a mensagem enfileirada.

O usuários presentes no canal é recuperado do dicionário *channels* e o usuário que está enviando a mensagem é filtrado para não recebê-la. Por fim, a lista de usuários presentes no canal é iterada e a mensagem do usuário é enviada.

3.2. Descrição do cliente

Assim como foi feita na classe do servidor, na classe de cliente também é criado um socket para realizar a conexão, são estabelecidos o ip e porta, por padrão 127.0.0.1 e 7777 respectivamente, há uma lista que irá guardar todas as mensagens digitadas por aquele usuário e um booleano para falar se a conexão com o servidor está ativa ou não.

```

class Client:
    server: socket.socket
    ip: str
    port: int
    input_list: list[socket.socket | TextIO]
    running: bool

    def __init__(self, ip, port):
        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.connect((ip, port))
        self.input_list = [sys.stdin, self.server]
        self.running = False

    def start(self):
        self.running = True
        self._run()

    def close(self):
        self.server.close()

    def _run(self):
        while self.running:
            inputs, _, _ = select.select(self.input_list, [], [])

            for sock in inputs:
                self._handle_input(sock)

    def _handle_input(self, socks):
        if socks == self.server:
            message = socks.recv(2048)
            try:
                if message:
                    print(message.decode("utf-8"), end='')
                return
            except Exception:
                ...
            self._exit_server()
        else:
            message = bytes(input(), 'utf-8')
            self.server.send(message)

    def _exit_server(self):
        print("Saindo do servidor...")
        self.input_list.remove(self.server)
        self.server.close()
        self.running = False

```

Figura 13: Configurações básicas do socket do cliente

O método `_run` faz com que, enquanto for verdadeira a propriedade do cliente que informa que ele está conectado com o servidor, o cliente recebe uma lista de sockets que tem dados pré-carregados e prontos para serem lidos. Há então um tratamento desses dados, caso o socket seja o mesmo utilizado na conexão com o servidor, significa que a mensagem deve ser recebida, mas caso o socket não seja o

citado anteriormente, então a mensagem deve ser escrita pelo usuário e enviada ao servidor.

4. Conclusão

O projeto de pesquisa apresentado conseguiu suprir a maioria das necessidades de comunicação em uma sala de bate papo, permitindo ao usuário entrar/criar/sair de uma sala, possuir identificação através de apelido (*nickname*) para facilitar a comunicação, permitir o usuário listar os canais existentes, e caso esteja em um canal, os usuários presentes no canal, e o mais importante que é a comunicação entre usuários da mesma sala. A principal limitação é a impossibilidade de comunicação utilizando áudio/vídeo, devido a dificuldade na implementação de uma interface web com a utilização de um websocket para realizar a comunicação entre cliente e servidor.

5. Experiência da equipe

- a. **Eduardo:** Acredito que a oportunidade de colocar em prática o conteúdo da disciplina sempre é um ponto positivo, porém acredito que atividades como essa deveriam ter sido feitas desde o início do semestre. Tive a oportunidade de aprender bastante e gostaria de que a conexão com o websocket tivesse funcionado. Quanto à participação, o grupo foi bem equilibrado em questão de debate de idéias, contribuição e esforço, todos colaboraram com os conhecimentos que tinham. Pra mim a nota de todos os membros do grupo deveria ser 10/10, comigo incluso.
- b. **Rafael:** O projeto fomentou a necessidade de uma atividade prática após muitas teóricas, e sinto que essa necessidade fez com que eu me esforçasse bastante resultando em um aprendizado incrível, tanto relacionado a sockets, quanto a syscall do select e todo o gerenciamento de sala de bate-papos. A participação do grupo foi praticamente igual devido ao fato de termos feito reuniões com pair programming, então todos se saíram muito bem e concordaram em distribuir a nota igualmente.
- c. **Thiago:** Gostei bastante do trabalho prático, tanto esse como o laboratório de DNS. Foi muito interessante conectar o que aprendi em sala com algo real e ver tudo isso funcionando é muito bom. A parte de fazer o servidor reconhecer a requisição HTTP e retornar um arquivo para ser mostrado no navegador foi bem interessante de se fazer, pois a princípio a interface do usuário seria pelo navegador, mas foi uma pena não ter conseguido realizar a conexão com o servidor por Websocket. O grupo trabalhou de forma bem homogênea, então a nota deveria ser distribuída igualmente.

6. Referências

SOCKET: Low-level networking interface. Python. Disponível em: <https://docs.python.org/3/library/socket.html>. Acesso em: 30 abr. 2022.

SELECT: *Wait for I/O Efficiently.* PyMOTW. Disponível em: <http://pymotw.com/2/select/#module-select>. Acesso em: 30 abr. 2022.

select(2). Linux manual page. Disponível em: <https://man7.org/linux/man-pages/man2/select.2.html>. Acesso em: 30 abr. 2022.

Cabeçalhos HTTP. MDN web docs. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Headers>. Acesso em: 01 maio de 2022.

HTTP Response. W3. Disponível em: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html>. Acesso em: 01 maio de 2022.