Relatório do Trabalho Prático

Redes de computadores 2
Aluno: Alan da Silva Gomes
Professora: Debora Christina Muchaluat Saade
Monitor: Rômulo Augusto Vieira Costa

Objetivo

Implementar um servidor socket usando o protocolo TCP/IP para o envio de mensagens de controle entre os usuários/clientes da rede.

Cada usuário se conecta ao servidor através do par (IP, PORTA) e através desse consegue enviar mensagens para os demais usuário conectados a esse mesmo servidor. Assim também como mensagens broadcast e solicitar a lista de informações de cada usuário conectado.

O que foi implementado

O servidor socket foi implementado e possibilita a troca de mensagens entre os usuários/sockets conetados.

- Listar os usuários conectados através de uma tabela dinâminca.
- Mandar mensagens broadcast.
- Verificar a unicidade dos usuários. Cada usuário possui um identificador único que é o seu nome. E esse atributo fica armazenado na tabela dinâminca.

Se faz a verificação da singularidade do nome do usuário, logo cada usuário deve possuir um nome diferente. Dessa forma não é possível haver mais um de usuário usando o mesmo nome, mesmo que a sua porta seja diferente.

- Listar o conteúdo de cada usuário. Por enquanto isso está sendo feito através de uma lista.

Implementação do código

Temos dois arquivos de códigos implementados na linguagem python3. Um para a implementação do servidor e outro para a implementação do cliente.

Implementação do código – Script server.py

server.py

```
1   import threading
2   import socket
3   import message as message_manager
4
5   # server setup
6   host = "localhost"
7   port = 1234
8   socket_server = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
9   socket_server.bind((host, port))
10  socket_server.listen(10)
11  BUFFER_SIZE = 1024
12  close_server = False
13  clients_connected = {}
14  """dictionary with all clients connected"""
```

A partir da linha 6 temos:

L6-L7 : configuração de um socket_server que usa um parte (IP,PORTA) = (localhost,1234)

L8: um objeto da classe socket da família Ipv4 e e tipo TCP

L9: ligação do socket em (localhost,1234)

L10: o socket irá escutar até 10 clientes.

L11: tamanho da mensagem enviada e recebida pela socket.

L12: variável de controle da execução do socket. Importante para conseguir fechar o socket quando solicitado.

L13: lista dinâmica com os clientes conectados.

Script server.py – receive()

A função receive() recebe as conexões do clients no servidor. Através dela o servidor escuta e conecta os clientes.

server.py

2 usages alangomes7

```
17  def receive():
18      """
19      receive client's connection: the server.
20      """
21      print("Server is running and listening...")
22      global close_server
23      while not close_server:
24          try:
25              # First: get client and client_name
26              socket_client, address = socket_server.accept()
27              # message to request client name
28              protocol_message = message_manager.protocol_message_encoding("server", "client", "name")
29              # using this method because the client is not added on clients_connect list yet
30              socket_client.send(protocol_message)
31              # receive client's name
32              received_message = message_manager.protocol_message_decoding(socket_client.recv(BUFFER_SIZE))
33              print(received_message)
```

L17 – L22: parâmetros e documentação da função
L23: loop que roda enquanto o servidor não for fechado.
L24 – L32: conexão do novo socket_client ao socket_server
L32: o server imprime a mensagem de conexão enviada pelo cliente

```
34              if username_available(received_message[3]):
35                  print("connection is established with: %s" % received_message[3])
36                  clients_connected[socket_client] = received_message[3]
37                  message_data = "You are now connected to the server"
38                  message_protocol = message_manager.protocol_message_encoding(
39                      "server", received_message[3], "connection_confirmation", message_data)
40                  message_manager.send_server_message(message_protocol, clients_connected)
41                  # now using threads to allow multiple connections and actions simultaneously
42                  thread = threading.Thread(target=handle_client, args=(socket_client,), daemon=True)
43                  thread.start()
44              else:
45                  message_data = "This username is already taken. "
46                  message_data += "You need to connect again and use another one"
47                  new_client_name = "new_client: " + received_message[3]
48                  protocol_message = message_manager.protocol_message_encoding("server", new_client_name,
49                                                                               "connection_error", message_data)
50                  socket_client.send(protocol_message)
51                  print(protocol_message)
52                  break
```

L34-L52: através do nome é verificado se o client já está conectado ou não no socket_server. Caso não esteja conectado, uma mensagem de confirmação de conexão é enviada para o cliente. E uma thread para gerenciar a conexão é criada.
Caso já esteja, uma mensagem de error de conexão é enviada para o cliente e o loop é encerrado.

```
53          except socket.error as socket_error:
54              if close_server:
55                  print("Server closed")
56              else:
57                  print(socket_error)
58                  print("Server closed")
```

L53-L58: Tratamento de erros.
L54-L55: Caso o server esteja fechado o valor da varável close_server é verdadeiro, logo o servidor imprime a mensagem “Server closed”. Aqui é gerado um erro para forçar o encerramento do server_socket e não é impresso na tela, pois o erro é intencional.
L56-L58: Caso o server seja fechado devido a algum erro de conexão do socket, o server imprime a mensagem de error e imprime a mensagem de “Server closed”. Aqui já é impresso o erro na tela.

Script server.py – handle_client(client)

```
61  def handle_client(client):
62      """
63      Manage client's messages
64      :param client: client sender
65      """
66      message_received = ["", "", "", ""]
67      global close_server
68      while not close_server:
69          try:
70              message_received = message_manager.protocol_message_decoding(client.recv(BUFFER_SIZE))
71              print("Server receive: %s" % message_received)
```

Gerencia as mensagens que o cliente envia para o servidor.
L61-L69: docuementação da função e variáveis de controle.
L70-L71: recebe a mensagem enviada pelo cliente e imprime-a

```
72              # server actions
73              if message_received[2] == "Unknown operation":
74                  message_data = "Operation (" + message_received[3] + ") is not supported"
75                  protocol_message = message_manager.protocol_message_encoding("server", message_received[0],
76                                                                               "Unknown_operation",
77                                                                               message_data)
78                  print("Server sends: %s" % protocol_message)
79                  message_manager.send_server_message(protocol_message, clients_connected)
80                  continue
81              if message_received[2] == "echo":
82                  protocol_message = message_manager.protocol_message_encoding(message_received[0], message_received[0]
83                                                                               "echo",
84                                                                               message_received[3])
85                  print("Server sends: %s" % protocol_message)
86                  message_manager.send_server_message(protocol_message, clients_connected)
87                  continue
88              if message_received[2] == "broadcast" or message_received[2] == "broadcast_not_me":
89                  protocol_message = message_manager.protocol_message_encoding(
90                      message_received[0], "all clients", message_received[2], message_received[3])
91                  print("Server sends: %s" % protocol_message)
92                  message_manager.send_server_message(protocol_message, clients_connected)
93                  continue
94              if message_received[2] == "list_clients" and message_received[1] == "server":
95                  protocol_message = message_manager.protocol_message_encoding("server", message_received[0]
```

L72-L139 (ver arquivo): lógica mais importante da função. Todas as ações que o servidor executa de acordo com a mensagem recebida.
As operações que o servidor faz são:

"Unknown operation": o usuário tentou executar ou digitou uma operação desconhecida pelo servidor. O servidor retorna a mensagem para o usuário e informa que a operação não é suportada.
"echo": o usuário executa uma mensagem para ser ecoada para ele mesmo, mesagem de eco.
"messagens broadcast": o servidor possui dois comportamentos no envio de mensagens broadcast.

"broadcast": informa as servidor que a mensagem deve ser enviada para todos, *inclusive* o cliente remetente.

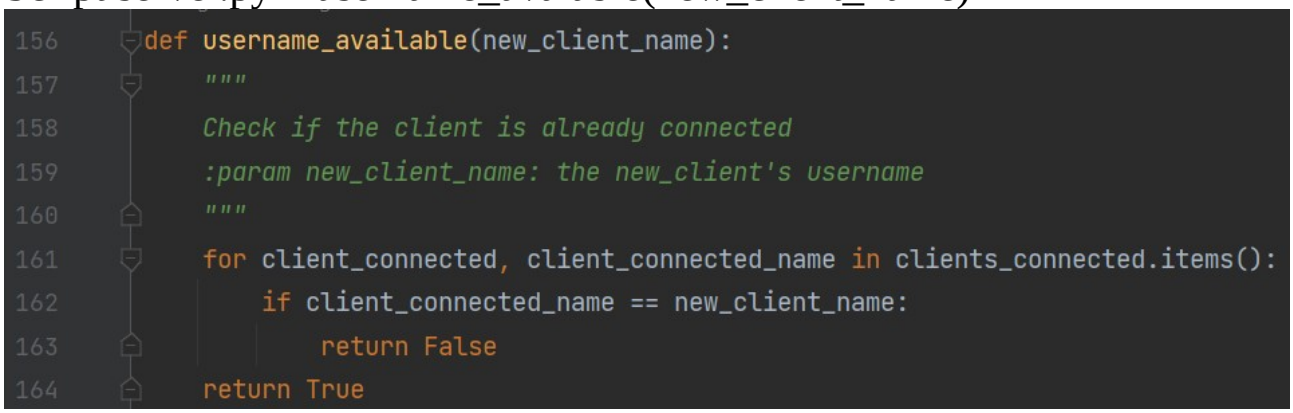
"broadcast_not_me": informa as servidor que a mensagem deve ser enviada para todos, *exceto* o cliente remetente.

"list_clients": lista todos os clientes conectados no servidor – lista a tabela dinâmica do servidor.
"list_files": lista a lista com os nome de aquivos de um cliente específico. Após digitar esse código é necessário informar de qual cliente deseja receber a lista com os nomes do arquivos.
"exit": informa ao servidor que o cliente deseja se desconectar. Após o servidor receber essa mensagem o cliente é removido da tabela dinâmica de clientes conectados e o cliente encerra a sua conexão. Esse comando encerra o programa do respectivo cliente.
"close_server": informar ao servidor para encerrar a sua conexão. O servidor então envia uma mensagem broadcast para todos os clientes conectados a ele. Então os clientes encerram as suas conexões e o servidor também encerra a sua conexão. Esse comando encerra o programa e o programa de todos os outros clientes conectados.

```
140         except socket.error as socket_error:
141             message_data = str(socket_error) + " | "
142             message_data += "client (" + message_received[0] + ") disconnected"
143             protocol_message = \
144                 message_manager.protocol_message_encoding(
145                     "server", "server", "error_message", message_data)
146             clients_connected.pop(client)
147             print("Server sends: %s" % protocol_message)
148             break
149     if message_received[2] == "close_server":
150         print("Closing server...")
151         socket_server.close()
152     else:
153         receive()
```

L140-L148: tratamento de exceções caso o cliente seja desconectado de forma abrupta ou inesperada. O servidor remove o cliente da tabela dinâmica de clientes conectados e imprime a mensagem informando da desconexão
L149-L151: caso o servidor tenha recebido a mensagem para o seu encerramento, então o socket_server é encerrado.
L151-153: caso o servidor esteja se recuperando de um erro, então a função receive() é invocada e o servidor continua a sua execução mantendo o seu estado atual.

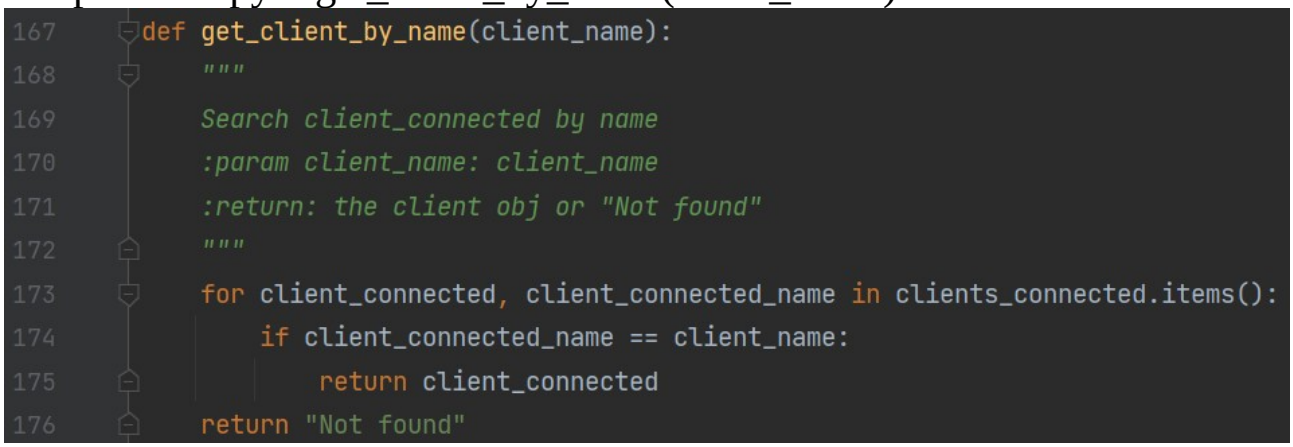
Script server.py – username_avalable(new_client_name)



```
156  def username_available(new_client_name):
157      """
158      Check if the client is already connected
159      :param new_client_name: the new_client's username
160      """
161      for client_connected, client_connected_name in clients_connected.items():
162          if client_connected_name == new_client_name:
163              return False
164      return True
```

Função que verifica através da lista de clientes conectados se o novo nome de usuário está disponível. Caso esteja, retorna verdadeiro, caso contrário retorna falso.
L156-160: parâmetros da função e sua documentação.
L161-164: lógica da função. Um laço *for* para verificar a existência do nome de usuário.

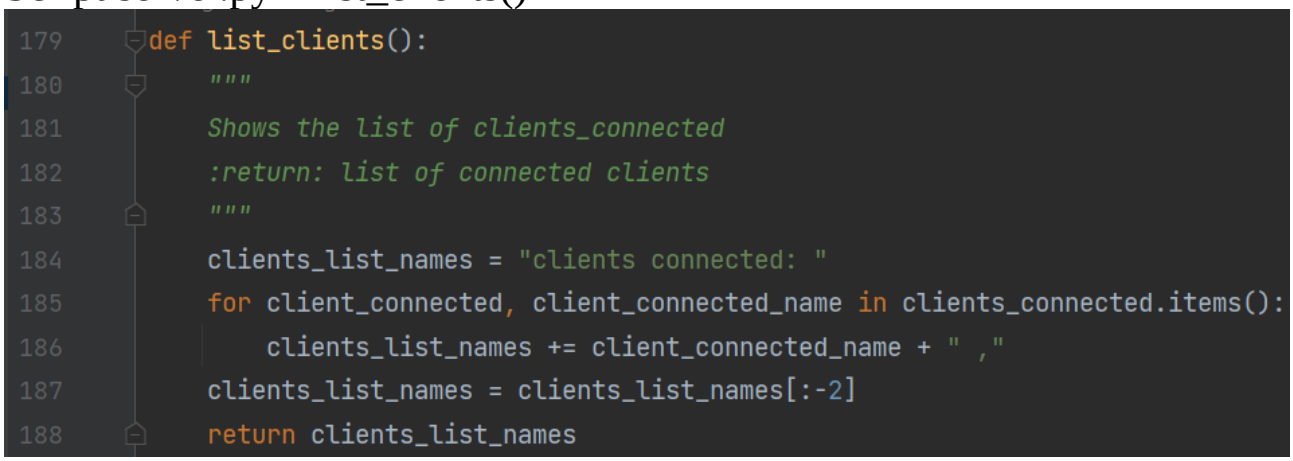
Script server.py – get_client_by_name(client_name)



```
167  def get_client_by_name(client_name):
168      """
169      Search client_connected by name
170      :param client_name: client_name
171      :return: the client obj or "Not found"
172      """
173      for client_connected, client_connected_name in clients_connected.items():
174          if client_connected_name == client_name:
175              return client_connected
176      return "Not found"
```

Função que retorna o usuário da lista de clientes conectados selecionando-o por nome.
L167-172: parâmetros da função e sua documentação.
L173-176: lógica da função. Um laço *for* para selecionar o usuário através do seu nome.

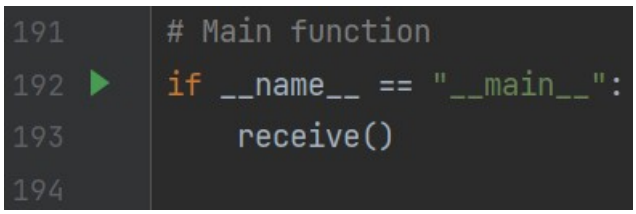
Script server.py – list_clients()



```
179  def list_clients():
180      """
181      Shows the list of clients_connected
182      :return: list of connected clients
183      """
184      clients_list_names = "clients connected: "
185      for client_connected, client_connected_name in clients_connected.items():
186          clients_list_names += client_connected_name + " ,"
187      clients_list_names = clients_list_names[:-2]
188      return clients_list_names
```

Função que lista os clientes conectados. Todos os clientes conectados as servidor se encontram na sua lista de clientes conectados.
L179-183: parâmetros da função e sua documentação.
L184-188: lógica da função. Um laço *for* para selecionar o usuário e inseri-lo na string de retorno

Função principal. Primeira parte do script que é executado para dar início ao programa.
192-193: chama a função receive() e inicia o servidor.



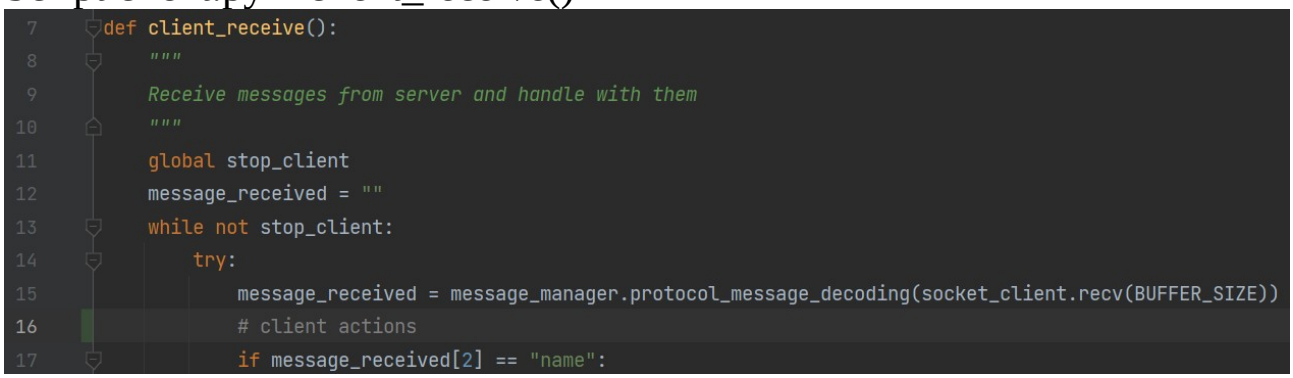
Implementação do código – Script client.py

Script client.py - main

```
118 if __name__ == "__main__":
119     # client's setup
120     name = str(input('write down your name: \n'))
121     # new socket, family: Ipv4, type: TCP
122     socket_client = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
123     host = "localhost"
124     port = 1234
125     socket_client.connect((host, port))
126     BUFFER_SIZE = 1024
127     stop_client = False
128     my_files = ["File 1", "File 2", "File 3"]
129
130     # receive thread
131     receive_thread = threading.Thread(target=client_receive)
132     receive_thread.start()
133
134     # send thread
135     send_thread = threading.Thread(target=client_send)
136     send_thread.start()
```

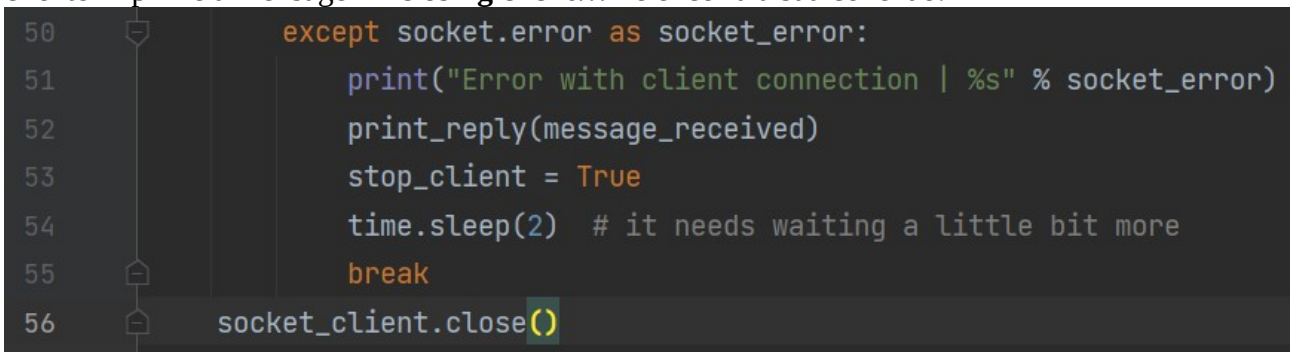
A parte desse *if* temos a execução do cliente.
L118-L128: parâmentros principais do cliente, sua documentação básica e váriavéis de controle.
L120: nome do cliente
L121-122: objeto do cliente e sua configurações. Um socket da família Ipv4 e tipo TCP.
L123-125: configurações de conexão do cliente. Um socket_client que usa um par (IP,PORTA) = (localhost,1234)
L126: tamanho da mensagem enviada e recebida pelo socket.
L127: variável de controle da execução do socket. Importante para conseguir fechar o socket quando solicitado.
L128: lista que será usada para listar a lista de nomes dos arquivos do cliente. Por enquanto estamos com uma lista pré-definida para todos os clientes.
L130-136: threads de execução do cliente.
L130-132: thread de recebimento de mensagens.
L134-136: thread de envio de mensagens.

Script client.py – client_receive()



```
7   def client_receive():
8       """
9       Receive messages from server and handle with them
10      """
11      global stop_client
12      message_received = ""
13      while not stop_client:
14          try:
15              message_received = message_manager.protocol_message_decoding(socket_client.recv(BUFFER_SIZE))
16              # client actions
17              if message_received[2] == "name":
```

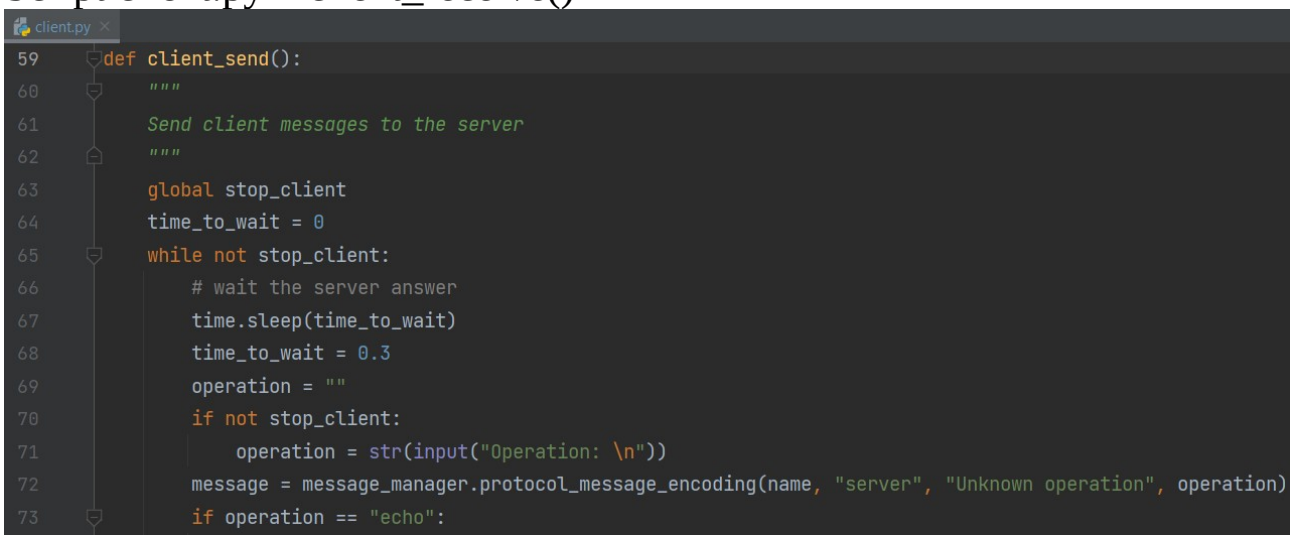
Função que implementa as operações do client de acordo com a mensagem recebida do servidor. Aqui vemos o cliente imprimindo as mensagens de acordo com a finalidade de cada mensagem.
L7-L14: parâmetros, variáveis de controle principais e documentação da função.
L13: a função se passa em um laço de repetição *while* que roda enquanto o cliente não for encerrado.
L15: mensagem recebida pelo cliente. De acordo com essa mensagem recebida do servidor o cliente executa diferentes operações.
L16-49 (ver arquivo): todas as operações que o cliente executa de acordo com a mensagem recebida.
As operações que o cliente faz de acordo com a mensagem recebida do servidor são:
"name": retorna o seu nome.
"Unknown_operation",**"echo"**,**"list_clients"**,**"broadcast_not_me"**,**"broadcast" e dado da mensagem igual a "server closed"**: quando o cliente recebe recebe alguma dessas mensagens o cliente apenas imprime a mensagem recebida.
"list_files": o cliente começa a trocar mensagens com o servidor para que consiga enviar e imprimir corretamente a mensagem com os arquivos dos solicitados.
"connection_error", **"exit"**, **"broadcast" e dado da mensagem igual a "server closed"**: o cliente imprime a mensagem **"closing client…"** e encerra a sua conexão.



```
50              except socket.error as socket_error:
51                  print("Error with client connection | %s" % socket_error)
52                  print_reply(message_received)
53                  stop_client = True
54                  time.sleep(2)  # it needs waiting a little bit more
55                  break
56          socket_client.close()
```

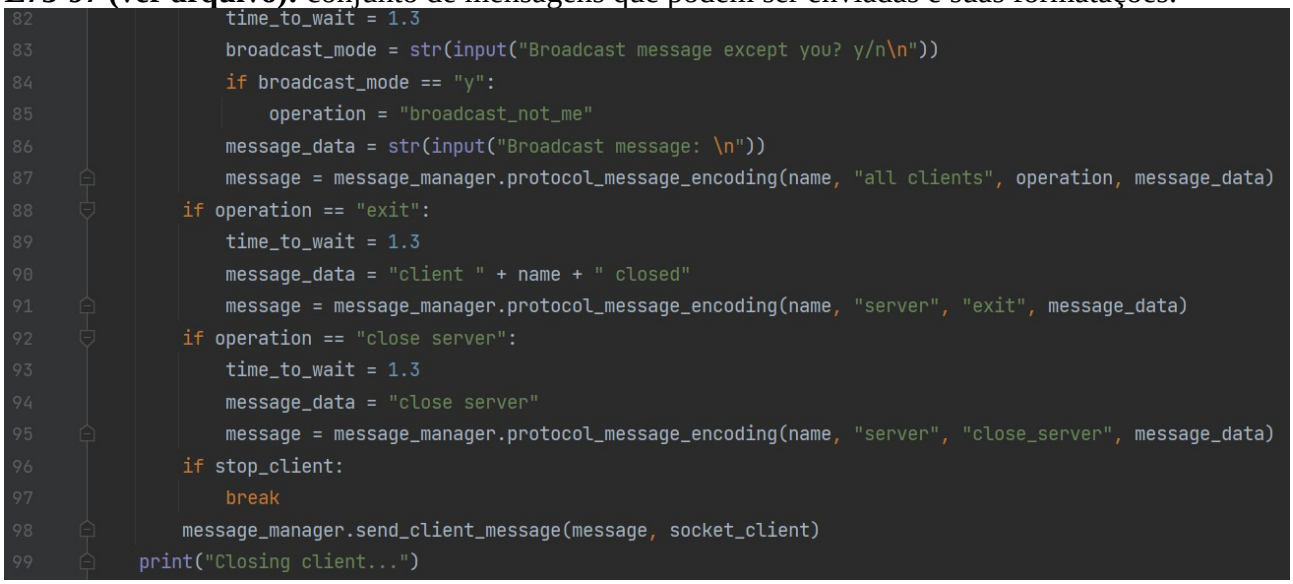
L50-56: tratamento de erros. O cliente imprime a mensagem informando que houve erros na sua execução e encerra o loop *for*.
L56: uma vez encerrado o loop *for* o cliente fecha o socket_client.

Script client.py – client_receive()



```
client.py
59  def client_send():
60      """
61      Send client messages to the server
62      """
63      global stop_client
64      time_to_wait = 0
65      while not stop_client:
66          # wait the server answer
67          time.sleep(time_to_wait)
68          time_to_wait = 0.3
69          operation = ""
70          if not stop_client:
71              operation = str(input("Operation: \n"))
72          message = message_manager.protocol_message_encoding(name, "server", "Unknown operation", operation)
73          if operation == "echo":
```

Função que recebe os dados digitados pelo usuário e os envia para o servidor. De acordo com cada tipo de operação solicitada é feito uma formatação de mensagem. As operações são as mesma citadas antes pelo recebimento do servidor.
O cliente fica em um laço *while* sempre esperando a próxima mensagem digitada pelo usuário para enviar ao servidor.
Operações enviadas pelo usuário: **"Unknown operation","echo","messagens broadcast","list_clients","list_files","exit"** e **"closer_server"**.
L59-L69: parãmetros da função, variáveis de controle e sua documentação.
L64: a variável time_to_wait informa o tempo em que o usuário deve aguardar para digitar a próxima mensagem. É um tempo importante, pois em algumas operações como “exit”, “close_service” e “broadcast” o servidor pode demorar mais tempo para processar a resposta.
L73-97 (ver arquivo): conjunto de mensagens que podem ser enviadas e suas formatações.



```
82              time_to_wait = 1.3
83              broadcast_mode = str(input("Broadcast message except you? y/n\n"))
84              if broadcast_mode == "y":
85                  operation = "broadcast_not_me"
86              message_data = str(input("Broadcast message: \n"))
87              message = message_manager.protocol_message_encoding(name, "all clients", operation, message_data)
88          if operation == "exit":
89              time_to_wait = 1.3
90              message_data = "client " + name + " closed"
91              message = message_manager.protocol_message_encoding(name, "server", "exit", message_data)
92          if operation == "close server":
93              time_to_wait = 1.3
94              message_data = "close server"
95              message = message_manager.protocol_message_encoding(name, "server", "close_server", message_data)
96          if stop_client:
97              break
98          message_manager.send_client_message(message, socket_client)
99      print("Closing client...")
```

L98: aqui temos o envio da mensage, já formatada pelas linhas anteriores.
L99: quando o usuário está encerrando esse laço *while* é firnalizado e o cliente imprime a mensagem **"Closing client…"** informando o fechamento do cliente.

Script client.py – print_reply()

```
5 usages  alangomes7
def print_reply(protocol_message_decoded):
    """
    Format the print to user
    :param protocol_message_decoded: message to extract the data to print
    """
    reply = "-----------------------------------------------------" + "\n"
    reply += "Reply: " + "\n"
    reply += "   Client sender: " + protocol_message_decoded[0] + "\n"
    reply += "   Client destination: " + protocol_message_decoded[1] + "\n"
    reply += "   Operation: " + protocol_message_decoded[2] + "\n"
    reply += "   Message: " + "\n"
    reply += "       " + protocol_message_decoded[3] + "\n"
    reply += "-----------------------------------------------------" + "\n"
    print(reply)
```

Função que formata a mensagem enviada pelo servidor para o cliente e então exibe-a para o usuário.
L102-106: parâmetros da função e sua formatação.
L107-115: lógica da função. Aqui o cliente formata a mensagem e imprime-a para o usuário.
L115: cliente imprime a mensagem formatada para o usuário.

Implementação do código – Script message.py

Script que codifica e decodifica as mensagens trocada entre o cliente e o servidor. Também, temos o roteamento de mensagens.

Script message.py – protocol_message_encoding()

```
message.py
23 usages  alangomes7
def protocol_message_encoding(x, y, fx, data="no data"):
    """
    Encode message str to bytes
    :param x: client sender
    :param y: client destination
    :param fx: message function
    :param data: message data
    :return: encoded message
    """
    message = str(x) + " | " + str(y) + " | " + str(fx) + " | " + str(data)
    return message.encode('utf-8')
```

Função que codifica a mensagem para enviar.
L1-L9: parâmetros da função e sua documentação.
L1: nos parâmtros da função temos um parâmetro *default* indicando que nem todas as mensagens precisam informar o campo de dado da mensagem. Temos mensagens que apenas enviam comandos.

L10: lógica da função. Transforma todos os argumentos passados em uma string.
L11: retorno dos bytes que representam a codificação da string.

Script message.py – protocol_message_decoding()

```
14  def protocol_message_decoding(protocol_message):
15      """
16      Decode message bytes to str
17      :param protocol_message: encoded message as bytes
18      :return: decoded message
19      """
20      message = str(protocol_message.decode('utf-8'))
21      return message.split(" | ")
22
```

Função que decodifica a mensagem enviada de bytes para string.
L14-19: parâmetros da mensagem e sua documentação.
L20: decodificação da mensagem de bytes para string.
L21: retorno da mensagem com formatação.

Script message.py - send_cliente_message()

```
24  def send_client_message(protocol_message, client):
25      """
26      Sends the message on client side
27      :param protocol_message: encoded message as bytes
28      :param client: client sender
29      """
30      client.send(protocol_message)
```

Função que realiza o envio da mensagem pelo cliente.
L24-29: parâmetros da mensagem e sua documentação.
L30: envio da mensagem.

Script message.py - send_server_message()

```
33  def send_server_message(protocol_message, clients_connected):
34      """
35      Sends the message on server side. This functions routes the message from clients in clients_connected_list
36      :param protocol_message: encoded message as bytes
37      :param clients_connected: The list of clients connected on server
38      """
39      protocol_message_decoded = protocol_message_decoding(protocol_message)
40      if protocol_message_decoded[2] == "broadcast_not_me" or protocol_message_decoded[2] == "broadcast":
41          send_server_message_broadcast(protocol_message, clients_connected)
42          return protocol_message
43      else:
44          forwarded_message = False
45          destination_client = protocol_message_decoded[1]
46          for client_connected, client_connected_name in clients_connected.items():
47              if client_connected_name == destination_client:
48                  client_connected.send(protocol_message)
49                  forwarded_message = True
50                  break
51          # reply message to sender when not found the client destination (echo)
52          if not forwarded_message:
53              message_data = "Destination client (" + destination_client + ") not found!"
54              protocol_message_reply = protocol_message_encoding("server", protocol_message_decoded[0],
55                                                                 protocol_message_decoded[2], message_data)
56              send_server_message(protocol_message_reply, clients_connected)
57              protocol_message = protocol_message_reply
58      return protocol_message
```

Função que realiza o envio da mensagem pelo servidor.
L33-38: parâmetros da função e sua documentação.
L33: parâmetro: protocol_message e clients_connected.
protocol_message: mensagem a ser enviada.
clientes_connected: lista de possíveis destinatários. Note que é através dessa função que o servidor realiza o encaminhamento de mensagens.
L39: decodificação da mensagem para extrair informações de roteamento.
L40-42: encaminhamento da mensagem em caso de mensagem broadcast.
L43-57: encaminhamento da mensagem de acordo com a rota informada na própria mensagem.
L44-50: encaminhamento da mensagem para o cliente encontrado na lista de clientes conectados.
L51-56: encaminhamento da mensagem de volta para o remetente informando que o destinatário não foi encontrada na lista de clientes conectados (messagem de eco).
L57: protocol_message é alterado com os dados da mensagem de resposta do destinatário não encontrado.
L58: retorno da protocol_message.

Script message.py - send_server_message_broadcast()

```
61  def send_server_message_broadcast(protocol_message, clients_connected):
62      """
63      Send Broadcast messages
64      mode: default-all users: any, all users except the sender : "broadcast_not_me"
65      :param protocol_message: message formatted as protocol
66      :param clients_connected: dictionary with clients connected
67      """
68      message = protocol_message_decoding(protocol_message)
69      mode = message[2]
70      client_name = message[0]
71      for client_connected, client_connected_name in clients_connected.items():
72          if client_connected_name == client_name and mode == "broadcast_not_me":
73              continue
74          client_connected.send(protocol_message)
```

Função que envia as mensagens de broadcast.
L61-67: parâmetros da função e sua documentação
L68: decodificação da mensagem para extrair informações de encaminhamento.
L69: tipo da mensagem de broadcast, mode. A partir desse valor o encaminhamento da mensagem pode ser feito em broadcast simples ou broadcast exclusivo.
Broadcast simples: envia a mensagem para todos, *inclusive* o remetente.
Broadcast exclusivo: envia a mensagem para todos, *excluindo* o rementente.
L71-74: lógica principal da função. Através do laço *for* na lista de clientes conectados a mensagem é enviada.
L72: encaminhamento ou não encaminhamento da mensagem de acordo com o mode.
L74: envio da mensagem.

Menssagem de protocolo (protocol_message)

A troca de mensagens nesse trabalho é feita utilizado o protocolo TR2.23 (Trabalho de Redes 2 2023). O formato da mensagem obrigatóriamente deve possuir: remetente, destinatário, operação. O último campo é o de dados, mas nem sempre é utilizado. O tamanho máximo da mensagem é definido pelo BUFFER_SIZE. Foi utilizado BUFFER_SIZE = 1024 bytes.

Menssagens como “close_server”, “list_clients” e outras nem sempre enviam dados de arquivo, mas todas elas devem possuir dados de controle. Abaixo temos o formato da mensagem:

REMETENTE	DESTINATÁRIO	OPERAÇÃO	DADOS = Opcional

Desafios

Já nos desafios:
- O encerramento da conexão do usuário ainda é um desafio por causa do uso das threads. Solucionei colocando variáveis globais e assim também extendi a funcionalidade para o servidor.
- Tratar os erros que são gerados quando o usuário é desconectado de forma abrupta sem encerrar o servidor, para que dessa forma, possibilitar que a conexão dos outros usuários não seja encerrada caso um deles seja desconectado.
- Encerrar a conexão com todos os usuários caso o servidor seja encerrado de forma abrupta ou através de uma mensagem enviada por um dos clientes. Logo, quando o servidor encerrar a conexão os outros usuários serão informados da queda do servidor e encerrão as suas conexões com servidor.

Próximos Passos

Implementar o envio de arquivos de áudio e possibilitar a sua reprodução em streaming. Para isso é necessário:
- Fornecer a lista de arquivos que cada usuário possui.
- Criar uma conexão UDP possibilitando o envio de datagramas.
- Implementar o reprodutor de áudio.

Ter uma interface de usuário mais interativa do que somente a de texto:
- Implementar uma interface gráfica.