

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 usuários 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 conectados.

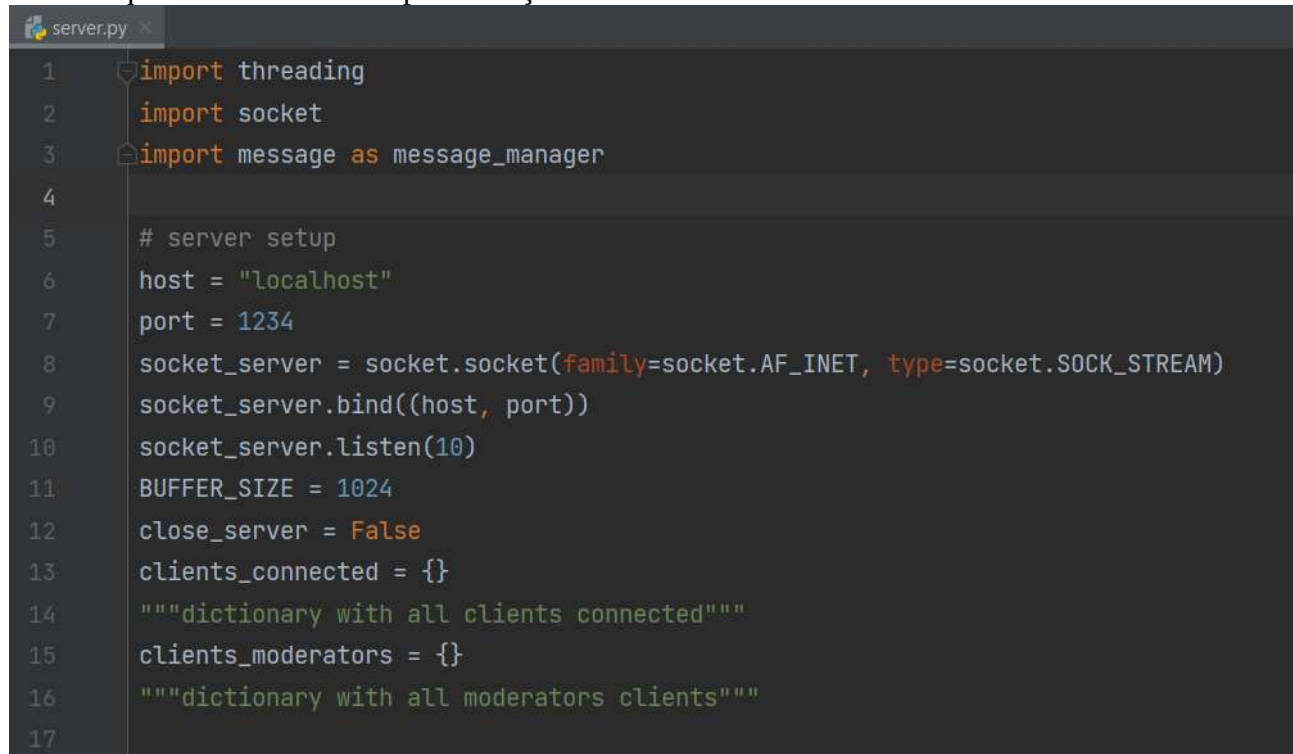
- Listar os usuários conectados por uma tabela dinâmica.
 - Mandar mensagens broadcast.
 - Verificar a unicidade dos usuários. Cada usuário possui um identificador único que é o seu nome. E esse atributo fica armazenado na tabela dinâmica.
- Se faz a verificação da singularidade através do nome do usuário, 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.
- Manda uma mensagem privada para um usuário específico.
 - O usuário consegue se desconectar.
 - O usuário consegue fechar o servidor.
 - O usuário consegue enviar uma mensagem de “eco”.

Implementação do código

Temos três arquivos de códigos implementados na linguagem python3. Um para a implementação do servidor, outro para a implementação do cliente e outro somente para o tratamento de mensagens (encaminhamento, codificação e decodificação).

Implementação do código – Script server.py

Nesse arquivo temos toda a implementação do servidor.



```
server.py x
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"""
15 clients_moderators = {}
16 """dictionary with all moderators clients"""
17
```

Aqui temos as configurações gerais do servidor. Um destaque vai para as linhas 13-16. Nessas linhas temos os dicionários que armazenam a tabela dinâmica de usuários conectados.

Também se pensou na possibilidade de restringir a operação de fechamento do servidor. Para isso será implementado uma verificação no dicionário de clientes moderadores. Apenas clientes listados nele poderão fechar o servidor.

Script server.py – receive()

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

```
server.py x
2 usages  alangomes7
19 def receive():
20     """
21     receive client's connection: the server.
22     """
23     print("Server is running and listening...")
24     global close_server
25     while not close_server:
26         try:
27             # First: get client and client_name
28             socket_client, address = socket_server.accept()
29             # message to request client name
30             protocol_message = message_manager.protocol_message_encoding("server", "client", "name")
31             # using this method because the client is not added on clients_connect list yet
32             socket_client.send(protocol_message)
33             # receive client's name
34             received_message = message_manager.protocol_message_decoding(socket_client.recv(BUFFER_SIZE))
35             print(received_message)
```

Nessa função é feita a verificação da lista de clientes por nome de usuário. Caso o nome de usuário já esteja em uso, logo esse usuário é desconsiderado e sua conexão é rejeitada.

Também foi necessário usar as threads. Através delas foi possível conectar mais de um usuário ao mesmo tempo.

```
server.py x
36 if username_available(received_message[3]):
37     print("connection is established: (%s) -> %s" % (received_message[3], address))
38     clients_connected[socket_client] = received_message[3]
39     # testar se o servidor possui alguém na lista de moderadores
40     message_data = "You are now connected to the server in " + str(address)
41     message_protocol = message_manager.protocol_message_encoding(
42         "server", received_message[3], "connection_confirmation", message_data)
43     message_manager.send_server_message(message_protocol, clients_connected)
44     print("Sent: " + message_protocol.decode('utf-8'))
45     # now using threads to allow multiple connections and actions simultaneously
46     thread = threading.Thread(target=handle_client, args=(socket_client,), daemon=True)
47     thread.start()
48 else:
49     message_data = "This username is already taken. "
50     message_data += "You need to connect again and use another one"
51     new_client_name = "new_client: " + received_message[3]
52     protocol_message = message_manager.protocol_message_encoding("server", new_client_name,
53                                                                     "connection_error", message_data)
54     socket_client.send(protocol_message)
55     print(protocol_message)
56     break
```

Caso o servidor seja fechado inesperadamente ele em seguida é reiniciado através do tratamento de erros. Aqui se preocupou ter o servidor sempre rodando caso algum cliente não tenha enviado a mensagem de fechamento.

```
57 except socket.error as socket_error:
58     if close_server:
59         print("Server closed")
60     else:
61         print(socket_error)
62         print("Server closed")
```

Script server.py – handle_client(client)

Gerencia as mensagens que o cliente envia para o servidor.

```
65 def handle_client(client):
66     """
67     Manage client's messages
68     :param client: client sender
69     """
70     message_received = ["", "", "", ""]
71     global close_server
72     while not close_server:
73         try:
74             message_received = message_manager.protocol_message_decoding(client.recv(BUFFER_SIZE))
75             print("Server receive: %s" % message_received)
76             # server actions
77             if message_received[2] == "Unknown operation":
78                 message_data = "Operation not supported!"
79                 protocol_message = message_manager.protocol_message_encoding("server", message_received[0],
80                                                                               "Unknown_operation",
81                                                                               message_data)
82                 print("Server sends: %s" % protocol_message)
83                 message_manager.send_server_message(protocol_message, clients_connected)
84                 continue
85             if message_received[2] == "echo":...
92             if message_received[2] == "private_message":
```

Aqui temos a variável global `close_server`. Através dela o programa encerra ou não as threads de tarefas do servidor.

Conforme o código da mensagem enviada pelo cliente, o servidor executa diferentes operações.

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, mensagem de eco.

"messages broadcast": o servidor possui dois comportamentos no envio de mensagens broadcast.

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

"broadcast_not_me": informa ao 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.

"private_message": envia uma mensagem privada somente para um usuário específico. Esse usuário específico o cliente que informa.

"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.

Para mais detalhes da implementação dessas operações ver o código.

```

128         except socket.error as socket_error:
129             message_data = str(socket_error) + " | "
130             message_data += "client (" + message_received[0] + ") disconnected"
131             protocol_message = \
132                 message_manager.protocol_message_encoding(
133                     "server", "server", "error_message", message_data)
134             clients_connected.pop(client)
135             print("Server sends: %s" % protocol_message)
136             break
137         if message_received[2] == "close_server":
138             print("Closing server...")
139             socket_server.close()
140         else:
141             receive()

```

Tratamento de erros. Através do tratamento de erros verificamos se ocorreu um erro. Caso ocorra a thread é imediatamente reiniciada. Mas também pode ser que a thread esteja sendo encerrada por algum cliente. Nesse caso, a thread encerra e o programa pode finalizar sem problemas.

Script server.py – username_available(new_client_name)

```

144 def username_available(new_client_name):
145     """
146     Check if the client is already connected
147     :param new_client_name: the new_client's username
148     """
149     for client_connected, client_connected_name in clients_connected.items():
150         if client_connected_name == new_client_name:
151             return False
152     return True

```

Verifica se o nome de usuário já existe fazendo a varredura na tabela de clientes conectados.

Script server.py – get_client_by_name(client_name)

```

1 usage  alangomes7
155 def get_client_by_name(client_name):
156     """
157     Search client_connected by name
158     :param client_name: client_name
159     :return: the client obj or "Not found"
160     """
161     for client_connected, client_connected_name in clients_connected.items():
162         if client_connected_name == client_name:
163             return client_connected
164     return "Not found"

```

Busca o cliente através de seu nome de usuário fazendo a varredura na tabela de clientes conectados.

```

167 def list_clients():
168     """
169     Shows the list of clients_connected
170     :return: list of connected clients
171     """
172     clients_list_names = "clients connected: "
173     for client_connected, client_connected_name in clients_connected.items():
174         clients_list_names += client_connected_name + " , "
175     clients_list_names = clients_list_names[:-2]
176     return clients_list_names

```

Script server.py – list_clients()

Lista todos os clientes que estão na tabela de clientes conectados.

Script server.py – list_clients()

```

179 # Main function
180 if __name__ == "__main__":
181     receive()

```

Executa o servidor.

Implementação do código – Script client.py

Script client.py - main

Configurações gerais do cliente.

```
131 if __name__ == "__main__":
132     # client's setup
133     name = str(input('write down your name: \n'))
134     # new socket, family: Ipv4, type: TCP
135     socket_client = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
136     host = "localhost"
137     port = 1234
138     socket_client.connect((host, port))
139     BUFFER_SIZE = 1024
140     stop_client = False
141
142     # receive thread
143     receive_thread = threading.Thread(target=client_receive)
144     receive_thread.start()
145
146     # send thread
147     send_thread = threading.Thread(target=client_send)
148     send_thread.start()
```

Nas configurações gerais do cliente temos o destaque para as linhas 142-148. Nessas linhas temos as duas threads responsáveis pela execução de mais de um cliente de forma simultânea.

Script client.py – menu()

```
99 def menu():
100     # Menu
101     options = "Options: \n"
102     options += "1 - echo\n"
103     options += "2 - list clients\n"
104     options += "3 - private message\n"
105     options += "4 - broadcast message\n"
106     options += "5 - exit\n"
107     options += "6 - close server\n"
108     print(options)
109     try:
110         user_answer = int(input("Input option's number:\n"))
111     except ValueError:
112         user_answer = 0
113     finally:
114         return user_answer
```

Esse é o menu que é exibido para o cliente. Nele temos todas as operações possíveis para os servidores tratar. Inclusive temos o tratamento de erros caso o cliente insira um valor inválido. Caso

isso aconteça a resposta do cliente (user_answer) é configurada para zero e o servidor pode tratar essa requisição como uma requisição inválida e retornar a mensagem de erro correspondente.

Script client.py – print_reply()

```
117 def print_reply(protocol_message_decoded):
118     """
119     Format the print to user
120     :param protocol_message_decoded: message to extract the data to print
121     """
122     reply = "\nIncoming message:\n"
123     reply += "    Client sender: " + protocol_message_decoded[0] + "\n"
124     reply += "    Client destination: " + protocol_message_decoded[1] + "\n"
125     reply += "    Operation: " + protocol_message_decoded[2] + "\n"
126     reply += "    Message: " + "\n"
127     reply += "        " + protocol_message_decoded[3] + "\n"
128     print(reply)
129
```

Todas as mensagens recebidas pelo cliente são formatadas antes de serem impressas. Dessa forma garantimos um mesmo padrão de exibição para o cliente.

Script client.py – client_send()

```
client.py x
47 def client_send():
48     """
49     Send client messages to the server
50     """
51     global stop_client
52     time_to_wait = 0.3
53     while not stop_client:
54         # wait the server answer
55         time.sleep(time_to_wait)
56         time_to_wait = 0.3
57     if not stop_client:
58         operation = menu()
59         match operation:
60             case 1: # echo message
61                 print("--- Sent an echo message ---\n")
62                 message_data = str(input("Echo message: \n"))
63                 message = message_manager.protocol_message_encoding(name, name, "echo", message_data)
64             case 2: # list clients
65                 print("--- Listing all clients connected ---\n")
66                 message = message_manager.protocol_message_encoding(name, "server", "list_clients")
67             case 3: # private message
```

Conforme a opção selecionada no menu, o cliente se encarrega de buscar todos os dados faltantes para o envio da mensagem e então envia a mensagem para o servidor. E o servidor, dependendo da mensagem, faz o encaminhamento ou tratamento da mensagem.

Para mais detalhes de todos os casos verificar o código.

Destacando as linhas 52 e 56 temos um tempo de espera em segundos para a thread que executa essa função. Esse tempo foi necessário para garantir que as mensagens do servidor cheguem ao lado do cliente a tempo. Um exemplo de desafio que foi resolvido foi: o menu aparecia antes da mensagem de confirmação enviada pelo servidor chegar.

```

93         if stop_client:
94             break
95         message_manager.send_client_message(message, socket_client)
96         print("Closing client...")

```

Fechamento do cliente. Caso o cliente seja encerrado uma mensagem informando o fechamento é impressa na tela do próprio cliente.

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":
18                message_manager.send_client_message(
19                    message_manager.protocol_message_encoding(name, "server", "name", name), socket_client)
20                continue
21            if message_received[2] == \
22                "Unknown_operation" or message_received[2] == "echo" or message_received[2] == "private_message" \
23                or message_received[2] == "broadcast_not_me" \
24                or message_received[2] == "connection_confirmation" \
25                or (message_received[2] == "broadcast" and message_received[3] != "server closed"):
26                print_reply(message_received)
27                continue
28            if message_received[2] == "list_clients":
29                print_reply(message_received)
30                continue

```

Recebe as mensagens enviadas até o cliente vindas do encaminhamento de servidor. As mensagens já chegam no formato do protocolo. Logo, o cliente também executa diferentes funções de acordo com o código da mensagem.

```

73         case 4: # broadcast
74             print("--- Sent a message to everyone on the Server ---\n")
75             time_to_wait = 1.3
76             broadcast_mode = str(input("Broadcast message except you? y/n\n"))
77             if broadcast_mode == "y":
78                 operation = "broadcast_not_me"
79                 message_data = str(input("Broadcast message: \n"))
80                 message = message_manager.protocol_message_encoding(name, "all clients", operation, message_data)

```

Um destaque para a mensagem de broadcast. Aqui é necessário especificar se a mensagem de broadcast será enviada também para o cliente remetente ou não. Caso seja, todos receberão a mensagem. Caso contrário, o cliente remetente não receberá a sua própria mensagem.

Em alguns casos foi necessário adicionar um atraso na execução do programa. Esse atraso foi inserido para que a sincronização do cliente e do servidor seja feita. Dependendo da mensagem o atraso aumenta para até 1.3 segundos ou permanece no padrão de 0.3 segundos.

Para mais detalhes de todos os casos verificar o código.

```

31         if message_received[2] == "connection_error" or message_received[2] == "exit" \
32            or (message_received[2] == "broadcast" and message_received[3] == "server closed"):
33             print_reply(message_received)
34             print("Closing client...")
35             stop_client = True
36             time.sleep(2) # it needs waiting a little bit more
37             continue
38         except socket.error as socket_error:
39             print("Error with client connection | %s" % socket_error)
40             print_reply(message_received)
41             stop_client = True
42             time.sleep(2) # it needs waiting a little bit more
43             break
44     socket_client.close()

```

Nessas linhas temos o fechamento do cliente por alguma razão. Como uma mensagem de fechamento do cliente precisa ser enviada para servidor, o servidor faz a exclusão do cliente do serviço e encerra a sua conexão com o cliente, adicionamos um atraso na execução do programa. Esse atraso pode ser verificado nas linhas 36 e 42.

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 x
23 usages  alangomes7

1  def protocol_message_encoding(x, y, fx, data="no data"):
2      """
3      Encode message str to bytes
4      :param x: client sender
5      :param y: client destination
6      :param fx: message function
7      :param data: message data
8      :return: encoded message
9      """
10     message = str(x) + " | " + str(y) + " | " + str(fx) + " | " + str(data)
11     return message.encode('utf-8')

```

Função que codifica a mensagem para enviar.

L1-L9: parâmetros da função e sua documentação.

L1: nos parâmetros 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 (mensagem 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 remetente.

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.

Mensagem de protocolo (protocol_message)

A troca de mensagens nesse trabalho é feita utilizando o protocolo TR2.23 (Trabalho de Redes 2 2023). O formato da mensagem obrigatoriamente 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.

Mensagens 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

- Trabalhar com as threads.

- O fechamento delas foi um grande desafio, visto que elas são necessárias para a execução de vários clientes ao mesmo tempo.

- A sincronização das threads do cliente e do servidor. Isso foi feito através de atrasos na execução do programa.

- O fechamento da thread do servidor e em seguida o fechamento das threads de todos os clientes. O servidor fechava e os clientes continuavam executando. Isso causava um erro de conexão. Agora, uma vez que o servidor é fechado, todos os clientes em seguidas recebem essa informação e se fecham.

Próximos Passos

- Implementar a conexão P2P.
- Ter uma interface de usuário mais interativa do que somente a de texto:
 - Implementar uma interface gráfica.
- Adicionar uma identificação para o tipo do cliente. A operação de fechamento do servidor, por enquanto, pode ser executada por todos os clientes. A ideia é que somente clientes com permissões avançadas (moderadores), possam encerrar o servidor.
- Adicionar a vídeos conferência.