Projeto 1 de Redes de Computadores
Implementação do Internet Relay Chat (IRC)

Leandro Beloti Kornelius, 21/1020900

Eduardo Marques Pereira, 21/1021004
Maylla Krislainy, 19/0043873

Janeiro de 2023

I. Introdução

Internet Relay Chat (IRC) é um protocolo para mensagens de texto na Internet em tempo real (chat) ou conferência síncrona.

O IRC é um tópico importante dentro da disciplina de redes de computadores porque foi uma das primeiras aplicações amplamente utilizadas da Internet para comunicação em tempo real. Permitiu a criação de salas de bate-papo, ou canais, onde vários usuários poderiam participar de discussões em tempo real.

Esse trabalho foi desenvolvido com o objetivo de compreender o desenvolvimento de aplicações cliente servidor e dos protocolos de rede, desenvolvendo uma aplicação IRC.

II. Fundamentação Teórica

A arquitetura cliente-servidor é uma maneira comum de projetar aplicativos em rede. Nessa arquitetura, um ou mais clientes interagem com um servidor central para acessar recursos ou serviços. Os clientes enviam solicitações ao servidor, que processa as solicitações e devolve uma resposta.

Sockets são uma interface de programação de baixo nível para criar conexões de rede. Eles permitem que os aplicativos enviem e recebam dados em uma rede usando um conjunto comum de protocolos de comunicação, como TCP e UDP.

O Wireshark é uma ferramenta de análise de protocolo de rede que permite capturar e inspecionar o tráfego de rede. Ele pode ser usado para solucionar problemas de rede e entender como diferentes protocolos e aplicativos estão se comunicando na rede.

O protocolo Internet Relay Chat (IRC) é um padrão amplamente utilizado para comunicação baseada em texto em tempo real na Internet. Ele permite que os usuários entrem em canais (salas de bate-papo virtuais) e se comuniquem entre si em tempo real. Um subconjunto do protocolo IRC seria uma versão simplificada do protocolo usado para esse aplicativo de bate-papo específico.

Para desenvolver o aplicativo de bate-papo, foi utilizado como linguagem de programação o Python para criar os componentes de servidor e cliente. Também foi utilizado usado a biblioteca de sockets para criar e gerenciar as conexões de rede e implementado o protocolo IRC para lidar com a comunicação entre os clientes e o servidor. E o Wireshark foi usado para testar e depurar a comunicação de rede.

III. Ambiente Experimental e Análise de Resultados

I. Descrição do Cenário

No ICP existem dois componentes principais: o servidor e seus clientes. Os múltiplos clientes realizam solicitações de comandos ao servidor. Ao receber o pedido de serviço, o servidor deve processar os comandos e executar a funcionalidade de cada um.

Tendo isso em vista, foram utilizados sockets para estabelecer a comunicação entre o servidor e seus clientes.

```python
import socket
ClientMultiSocket = socket.socket() # inicializa o socket do cliente
host = '127.0.0.1'
port = 2004

# Avisa tentativa de conexao e alerta erro da biblioteca caso n efetivada
print('Waiting for connection response')
try:
    ClientMultiSocket.connect((host, port))
except socket.error as e:
    print(str(e))

entrada = ClientMultiSocket.recv(1024).decode() # Recebe confirmacao do server
print(entrada)

# Loop de entrada do cliente para enviar ao server os comandos e ter processamentos
while entrada != "QUIT":
    entrada = input('>> ')
    ClientMultiSocket.send(str.encode(entrada)) # Envia comando ao server
    resposta = ClientMultiSocket.recv(1024).decode() # Recebe processamento do servidor
    print('Response from server:\n' + resposta)

# Processa QUIT (necessario para remover cliente das estruturas)
ClientMultiSocket.send(str.encode(entrada)) # Envia comando QUIT ao server
resposta = ClientMultiSocket.recv(1024).decode() # Recebe resposta
print('Response from server:\n' + resposta)

# Finaliza socket do cliente
ClientMultiSocket.close()
```

Com isso, é possível observar como o cliente é instanciado usando a biblioteca socket do python, conectado ao servidor e ter suas instruções enviadas para o servidor.

```python
import ...

# Inicializacao do servidor e algumas variaveis
ServerSideSocket = socket.socket()
host = '127.0.0.1'
port = 2004
ThreadCount = 0
cond_parada = False
cond = True

'''
Estrutura dos dicionarios definidos
------------------------------------------------------------
Estrutura address
dic addresses -> chave address, valor: [nick, conn]
Estrutura cliente
dic clientes -> chave: nick, valor: [realname, host, port]
Estrutura canais
dic canais -> chave: nomecanal, valor: [nicks]
'''

# Inicializacao dos dics colocando alguns canais para teste
addressclientes = {}
clientes = {}
canais = {"CANAL1": [], "CANAL2": [], "CANAL3": []}

# Tenta associar host a porta
try:
    ServerSideSocket.bind((host, port))
except socket.error as e:
    print(str(e))
print('Socket is listening..')
ServerSideSocket.listen(5) # Quantidade de clientes possiveis
```

O servidor deve armazenar algumas informações como: conexões com os clientes para enviar o processamento dos comandos, as informações pessoais dos clientes e os canais com os devidos participantes. Semelhante aos clientes, o servidor deve ser instanciado e associada a um host e port.

Estrutura dos dicionários que armazenam as informações:

- O primeiro dicionário denominado *addressclientes* tem como chave o endereço da conexão do usuário e uma lista que armazena o nickname da conexão no primeiro elemento e a conexão no segundo.

- O segundo dicionário, clientes, armazena as informações de cada cliente, por isso tem como chave o nickname e uma lista associada contendo o nome real do cliente, o host do cliente e a porta do cliente.

- Por fim, o terceiro dicionário, canais, contém os clientes de cada canal. Ou seja, tem como chave o nome de cada canal e uma lista com os nicknames que pertencem a este canal.

Após isso, aguarda solicitações dos clientes e processa os devidos comandos. Cada comando tem sua própria estrutura e retorno para o usuário que será abordado abaixo.

```python
def whoChannelHandler(canal, dicCanais):  # WHO
    retorno = ""
    if canal in dicCanais.keys():
        retorno += f'Usuarios do {canal}:\n'
        cont_client = 1
        temalguem = False
        for cliente in dicCanais[canal]:
            temalguem = True
            retorno += f'{cont_client} - {cliente}\n'
            cont_client += 1
        if not temalguem:
            retorno += "Nao ha users nesse canal"
        return retorno
    return "Canal n existente"
```

```python
        # Demais casos, comando invalido ou tentativa de mensagem
        else:
            if data[0] == "VISUALIZAR":
                continue
            else:
                data = "Comando invalido"

        # Testes para visualizar comando mudando estruturas definidas
        print(addressclientes)
        print(clientes)
        print(canais)

        connection.send(data.encode()) # Envia processamento de volta ao cliente
```

I.1 Comandos

O comando NICK altera o nickname dos clientes. Em função da coleção de dados estabelecida anteriormente o nickname é único e, por consequência, para verificar a existência de outro usuário com o mesmo nickname basta olhar nas chaves do dicionário clientes. Não tendo ninguém com o nickname solicitado basta alterar todas ocorrências do nickname nos dicionários definidos.

```python
while True:
    Client, address = ServerSideSocket.accept() # Aceita conexao

    # Inicializa cliente com nicks arbitrarios e os adiciona nos dics
    if cond:
        nome_temp = 0
        cond = False
    addressclientes[address] = [str(nome_temp), Client]
    clientes[str(nome_temp)] = ["realnameinicial", '127.0.0.1', 2004]
    nome_temp += 1

    # Confirmacao de conexao do cliente no server
    print('Connected to: ' + address[0] + ':' + str(address[1]))
    start_new_thread(multi_threaded_client, (Client, ))
    ThreadCount += 1
    print('Thread Number: ' + str(ThreadCount))
```

Em virtude do cliente também ter o nome real dele, criamos o comando NAME com intuito de deixar que os clientes alterem o seu nome real.

```python
def conn_user(user, dicAddresses):
    for address in dicAddresses.keys():
        if dicAddresses[address][0] == user:
            return dicAddresses[address][1]


def address_conn(conn, dicAddresses):
    for adress in dicAddresses.keys():
        if dicAddresses[adress][1] == conn:
            return adress
```

O cliente deve conseguir verificar as suas informações. Nesse sentido, o comando USER apresenta ao usuário todas as suas informações relevantes como nickname, nome real, host e porta.

```python
def subscribeChannelHandler(address, canal, dicAddresses, dicCanais):  # JOIN
    usuario = dicAddresses[address][0]
    if canal not in dicCanais.keys():
        return 'O canal nao existe'
    for channel in dicCanais.keys():
        if usuario in dicCanais[channel]:
            unsubscribeChannelHandler(address, channel, dicAddresses, dicCanais)
    dicCanais[canal].append(usuario)
    return f'{usuario} foi adicionado ao {canal}'


def unsubscribeChannelHandler(address, canal, dicAddresses, dicCanais):  # PART
    usuario = dicAddresses[address][0]
    if canal not in dicCanais.keys():
        return 'O canal nao existe'
    elif usuario not in dicCanais[canal]:
        return 'O cliente nao esta no canal'
    dicCanais[canal].remove(usuario)
    return f'{usuario} foi removido do {canal}'
```

Como comando finalizador da conexão entre o cliente e o servidor, o QUIT deve finalizar o loop que permite ao cliente mandar solicitações de comando e remover todas ocorrências do cliente nas estruturas de armazenamento definidas. Além disso, deve ocorrer um aviso aos demais integrantes do canal que o cliente saiu.

Por isso, é realizado um loop que envia esta mensagem para todos os participantes que compartilhavam o canal. Para facilitar em enviar mensagens para múltiplas conexões, foi estabelecida a função *conn user*. Nesse aspecto, observa-se abaixo que ao receber o nickname e o dicionário dos endereços a função retorna a conexão do usuário identificado pelo nickname.

```python
def listChannelHandler(dicCanais):  # LIST
    retorno = ""
    for canal in dicCanais.keys():
        retorno += f'{canal}:\n'
        cont_client = 1
        if dicCanais[canal] != []:
            for clientes in dicCanais[canal]:
                retorno += f'{cont_client} - {clientes}\n'
                cont_client += 1
        else:
            retorno += 'Canal sem clientes vinculados.\n'
    return retorno
```

```python
def nameClientHandler(address, name_novo, dicAddresses, dicClientes):  # NAME
    nickname = dicAddresses[address][0]
    dicClientes[nickname][0] = name_novo
    return "Operacao concluida"


def userClientHandler(address, dicAddresses, dicClientes):  # USER
    usuario = dicAddresses[address][0]
    if usuario in dicClientes.keys():
        return f'Nickname: {usuario}\nName: {dicClientes[usuario][0]}\nHost: {dicClientes[usuario][1]}\nPorta: {dicClientes[usuario][2]}'
```

Ao especificar o canal que se deseja ser incluído, o comando JOIN insere o cliente ao canal especificado. Entretanto, o canal pode não existir ou o cliente pode já pertencer a outro canal. Para verificar a existência do canal é verificado a existência do mesmo nas chaves do dicionário clientes. Se o canal existir deve ocorrer a remoção do nickname da lista daquele canal em que a definição da função do comando PART foi reaproveitada. Em seguida basta adicionar o nickname na lista dos participantes do canal.

```python
def nickClientHandler(address, nickname_novo, dicAddresses, dicClientes, dicCanais):  # NICK
    nickname_velho = dicAddresses[address][0]
    if nickname_novo == nickname_velho:
        return "Nick especificado ja e o seu"
    elif nickname_novo in dicClientes.keys():
        return "Nickname indisponivel"
    else:
        dicAddresses[address][0] = nickname_novo
        for canal in dicCanais.keys():
            if nickname_velho in dicCanais[canal]:
                ind = dicCanais[canal].index(nickname_velho)
                dicCanais[canal][ind] = nickname_novo
        info_nickname = dicClientes[nickname_velho]
        del dicClientes[nickname_velho]
        dicClientes[nickname_novo] = info_nickname
        return "Operacao concluida"
```

O comando PART remove o cliente do canal especificado. Para isso, deve ocorrer duas verificações: a existência do canal e o pertencimento do cliente. Ambos problemas são verificados na função abaixo retornando ao cliente qual das verificações falhou. Se ambas condições forem validadas ocorre a remoção do nickname da lista daquele canal.

```python
def privMsgChannelHandler(address, entrada, msg, dicAddresses, dicClientes, dicCanais):  # PRIVMSG
    user_origem = dicAddresses[address][0]
    if entrada in dicCanais.keys():
        msg = f'Mensagem recebida pelo {user_origem} para o canal {entrada} -> ' + msg
        users_canal = dicCanais[entrada]
        for user in users_canal:
            if user != user_origem:
                conn = conn_user(user, dicAddresses)
                conn.send(msg.encode())
        return f'Mensagem enviada para o canal {entrada}.'
    elif entrada in dicClientes.keys():
        msg = f'Mensagem recebida pelo {user_origem} -> ' + msg
        conn = conn_user(entrada, dicAddresses)
        conn.send(msg.encode())
        return f'Mensagem enviada para o user {entrada}.'
    return "O user ou canal digitado n existe"
```

Para possibilitar que os clientes visualizem quem está em cada canal, o comando PART habilita essa visão. Sob essa análise, o comando retorna uma mensagem padrão para cada canal especificando o nome do canal e, se houver clientes presente no canal, lista o nome de cada. Caso contrário, especifica que não há clientes no canal.

```python
def quitHandler(address, dicAddresses, dicClientes, dicCanais):  # QUIT
    usuario = dicAddresses[address][0]
    if usuario in dicClientes.keys():
        del dicAddresses[address]
        del dicClientes[usuario]
        for canal in dicCanais.keys():
            if usuario in dicCanais[canal]:
                dicCanais[canal].remove(usuario)
                break
        for user in dicCanais[canal]:
            msg = f'O user {user} saiu do canal {canal}'
            conn = conn_user(user, dicAddresses)
            conn.send(msg.encode())
        return "Desconectando..."
```

O comando PRIVMSG deve habilitar que o usuário se comunique individualmente com outros clientes ou com o grupo inteiro a que pertence. Sob essa ótica, recebe um nome que pode ser de um canal ou outro cliente. Caso seja de canal, a função *conn user* é utilizada novamente para poder obter a conexão dos clientes presentes no canal. Para a mensagem digitada pelo cliente não seja mandada pra ele mesmo é utilizado uma condição. Sendo o nome digitado

um cliente é enviada a mensagem para o usuário especificado. Por fim, caso não exista o nome especificado nos canais e clientes é reportado o erro a quem fez a solicitação.

```
147  def whoChannelHandler(canal, dicCanais):  # WHO
148      retorno = ""
149      if canal in dicCanais.keys():
150          retorno += f'Usuarios do {canal}:\n'
151          cont_client = 1
152          temalguem = False
153          for cliente in dicCanais[canal]:
154              temalguem = True
155              retorno += f'{cont_client} - {cliente}\n'
156              cont_client += 1
157          if not temalguem:
158              retorno += "Nao ha users nesse canal"
159          return retorno
160      return "Canal n existente"
```

Semelhante ao comando LIST, a função WHO especifica os usuários presente no canal especificado. Logo, é necessário validar a existência do canal enviado e, após isso, retornar as informações conforme a estrutura abaixo. Caso não exista alguém no canal especificado, isso é avisado ao cliente que fez a solicitação.

II. Análise de Resultados

Dada a proposta do trabalho pela professora, buscamos entender o que seria necessário ser feito, e quais ferramentas utilizaríamos para a produção do que foi proposto.

O primeiro ponto a ser discutido e definido seria a linguagem de programação que seria utilizada para a produção do código dos programas. A linguagem deveria ter funções e bibliotecas que fornecessem suporte as necessidades que definimos anteriormente, além disso, seria de prioridade que todos os membros do grupos possuíssem algum nível de conhecimento prévio e experiência com a linguagem. Através desses critérios estabelecidos, foi decidido em discussão que utilizaríamos Python como linguagem de programação que seria utilizada para a produção do projeto proposto.

Inicialmente, decidimos por primeiro realizar a produção dos métodos requisitados na descrição do projetos, seriam os métodos básicos, métodos do servidor e métodos avançados. Cada membro do grupo ficou responsável por um método básico e um método do servidor, já os métodos avançados, foi decidido que faríamos em conjunto após a conclusão dos métodos antecessores.

Após a conclusão da produção de todos os métodos requisitados, partimos para a produção da estrutura do cliente, e em seguida realizamos a produção da estrutura do servidor, juntamente com a estrutura responsável pela execução e integração dos programas. Realizamos a produção do Cliente e Servidor em conjunto através de reuniões frequentes no Discord e discussões a todo momento pelo WhatsApp.

Neste ponto o código estava funcional, conseguimos testar os métodos e armazenar o que era feito, porém chegamos no primeiro impasse, embora o código estivesse 'completo', não conseguíamos realizar a conexão simultânea com mais de um socket.

A partir dessa limitação, buscamos todas as soluções possíveis. Encontramos a biblioteca thread, que seria a solução ideal para o nosso problema, estudamos e entendemos a sua implementação e partimos para a inserção dos métodos que permitiriam a conexão simultânea de vários sockets com o servidor. Após algumas tardes de testes e modificações nos programas, em fim conseguimos a conexão multithread em nosso projeto.

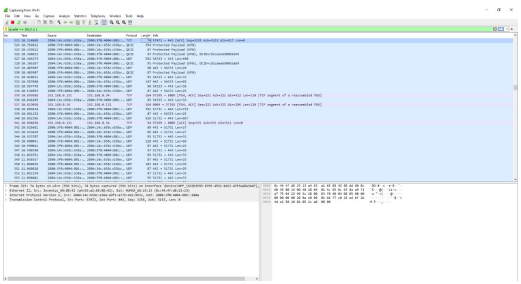
A conclusão do projeto se deve por conta do trabalho em conjunto de todo o grupo, todos se disporam a iniciar e concluir o projeto, a comunicação foi fundamental.

A experiência obtida na produção do projeto proposto, será de extrema importância durante toda as nossas respectivas trajetórias como profissionais. Cada um em seu individual, obteve uma nova perspectiva do que se pode ser feito com o que foi aprendido.

III. Wireshark

Em anexo o link para um vídeo de demonstração do funcionamento do servidor.
Video Aqui -> LINK

Quadro 1



IV. Conclusão

Concluindo, o trabalho apresentado é uma implementação server-side de um chat na Internet usando um subconjunto do protocolo Internet Relay Chat (IRC) em Python. O código usa as bibliotecas *socket* e *thread* para lidar com conexões de rede e várias conexões simultaneamente.

Akem disso foram ultilizados dicionários para armazenar e gerenciar informações de cliente, canal e endereço e funções para lidar com diferentes comandos de IRC. Também utilizamos um bloco try-except para lidar com erros ao vincular o host à porta.

O grupo como um todo parece ter tido um bom entendimento dos conceitos e técnicas utilizadas neste projeto. O uso de Python como linguagem de programação, biblioteca de soquetes e biblioteca de encadeamento foram escolhas

que acreditamos terem sido apropriadas, e a capacidade da equipe de trabalhar em conjunto e se comunicar de forma eficaz foi a chave para a conclusão bem-sucedida do projeto.

No geral, parece que o projeto atingiu nossos objetivos e foi implementado de acordo com os requisitos estabelecidos na descrição do projeto. A habilidade da equipe em superar desafios, como a conexão simultânea de vários soquetes, e a atenção aos detalhes na implementação do protocolo IRC são satisfatórias.

V. Referencias

Para o desenvolvimento do trabalho foi utilizado como forma de consulta os seguintes conteúdos como referência bibliográfica:

author = "Gordon McMillan",
title = "Socket Programming HOWTOű",
url = "https://docs.python.org/3.7/howto/sockets.html"

author = "Gabriel Ferreira", title = "Redes de Computadores: Exemplo IRC",
url = "https://github.com/Gabrielcarvfer/Redes-de-Computadores-UnB/blob/master/trabalhos/20181/Lab2/ExemploIRC.py"

author = "Pankaj",
title = "Python Socket Programming - Server, Client Example",
url = "https://www.digitalocean.com/community/tutorials/python-socket-programming-server-client"

author = "Lucas Teixeira",
title = "Tutorial de Utilização Básica do Software Wireshark",
url = "https://www.youtube.com/watch?v=TYk6ejP7dmI"

author = "Stack Overflow",
title = "server.listen(5) vs multithreading in socket programming",
url = "https://stackoverflow.com/questions/53880028/server-listen5-vs-multithreading-in-socket-programming"

author = "Digamber",
title = "How to Create Socket Server with Multiple Clients in Python",
url = "https://www.positronx.io/create-socket-server-with-multiple-clients-in-python/"

author = "Stack Overflow",
title = "Sockets: Simulate multiple clients in python using one machine/NIC",
url = "https://stackoverflow.com/questions/10901249/sockets-simulate-multiple-clients-in-python-using-one-machine-nic"

author = "Stack Overflow",
title = "WinError 10061, Python Server and Client connection problem",
url = "https://stackoverflow.com/questions/63911712/winerror-10061-python-server-and-client-connection-problem"

author = "Stack Overflow",
title = "Errno 10061 : No connection could be made because the target machine actively refused it (client - server)",
url = "https://stackoverflow.com/questions/12993276/errno-10061-no-connection-could-be-made-because-the-target-machine-actively-re"