Projeto 01
Teleinformática e Redes 2

Priscila Angel - 190094699, Rafael Rodrigues - 190134780, Victor Hugo - 190129794

***Resumo*—Este projeto objetiva consolidar e pôr em prática os conhecimentos sobre a camada de aplicação adquiridos na disciplina de Teleinformática e Redes 2. Consiste em uma aplicação que implementa um subconjunto do protocolo Internet Relay Chat (IRC), um sistema de chat que utiliza a arquitetura cliente-servidor, por meio da utilização de sockets.**

***Palavras-chave*—socket, cliente-servidor, TCP, IRC.**

I. Introdução

O Trabalho consiste em desenvolver uma aplicação de chat para uso local, utilizando o protocolo TCP (Transmition Control Protocol) e IRC (Internet Relay Chat), o protocolo TCP tem por finalidade manter a conexão confiável entre o cliente e o servidor e o IRC é um protocolo que permite a comunicação de texto em tempo real e é utilizado para conversação em grupos de chat. A finalidade é criar uma aplicação cliente-servidor, simulando diferentes condições de conexão. Para isso, será necessário criar um socket em uma linguagem de programação para estabelecer a conexão entre as partes. Além disso, utilizaremos o programa Wireshark para realizar testes e assim aprender mais sobre a camada de aplicações.

Janeiro 17, 2023

II. Fundamentação Téorica

A. Socket

Um socket é um mecanismo que permite a comunicação entre programas através da rede, ele é utilizado como ponto final para o envio e recebimento de dados. Ele é uma interface de software que utiliza protocolos de rede específicos, como TCP ou UDP, para permitir que os programas envie e receba dados. Cada socket é identificado por um endereço IP e um número de porta, onde o endereço IP é usado para identificar o dispositivo na rede e o número de porta é usado para identificar um processo específico no dispositivo. Sockets possibilitam que os programas se comuniquem de forma transparente através da rede independentemente do sistema operacional ou plataforma de hardware.

B. TCP

TCP (Transmission Control Protocol) é um protocolo de comunicação de rede que garante a confiabilidade na transmissão de dados entre dispositivos conectados em uma rede. Ele faz parte da pilha de protocolos da internet, junto com o IP (Internet Protocol). O TCP é um protocolo orientado à conexão, o que significa que ele estabelece uma conexão confiável entre dispositivos antes de qualquer troca de dados. Ele também é responsável por verificar e corrigir erros, controlar o fluxo de dados e retransmitir pacotes perdidos, garantindo que os dados sejam entregues corretamente e na ordem certa para o destinatário. Assim o TCP se torna uma opção viável para aplicações que tem necessidade de ter certeza da chegada dos dados.

C. IRC

O protocolo de comunicação de texto em tempo real chamado IRC permite conversas entre usuários de diferentes dispositivos e redes. Ele foi criado em 1988 e é usado principalmente para chats em grupo e comunicação instantânea. Os usuários se conectam a um servidor específico e se juntam a canais para conversar com outros usuários. Ele é amplamente utilizado em comunidades de jogos online, projetos de código aberto e grupos de discussão.

D. Python

Linguagem de programação que nos permite o desenvolvimento da aplicação por meio de linhas de código, para melhor utilização fizemos a importação da biblioteca de Sockets e utilizamos das ferramentas da linguagem como suas estruturas de dados e suas funções pré-programadas, além disso, o python possui uma biblioteca prompt toollkit que possibilita a criação de algumas ferramentas para o terminal deixando-o mais prático de ser utilizado.

III. Ambiente Experimental e Análise de Resultados

A. Descrição do Cenário

As observações dos resultados foram realizadas por meio de um notebook rodando o sistema operacional Ubuntu 20.04.3 LTS e um celular Android. Foi utilizada a linguagem de programação Python na sua versão 3 por ter sido a linguagem na qual foi disponibilizado o material base do projeto[1], além de sua simplicidade e do conhecimento prévio da equipe. A única biblioteca externa utilizada foi a prompt_toolkit[2], usada no cliente para personalizar a interface do terminal. A topologia da rede utilizada é mostrada na figura 1. O notebook é utilizado como servidor IRC e também pode rodar múltiplos clientes. Para executar o cliente no celular, foi utilizado o programa Pydroid 3[3], um interpretador e IDE de Python para celular.

[1] https://github.com/Gabrielcarvfer/Redes-de-Computadores-UnB
[2] https://python-prompt-toolkit.readthedocs.io/en/master/
[3] https://play.google.com/store/apps/details?id=ru.iiec.pydroid3

O mesmo código do cliente que roda no computador pode ser executado no celular sem alterações. Foi escolhida essa abordagem ao invés de criar um aplicativo de cliente para celular, pois nenhum membro da equipe possui conhecimento para tal.

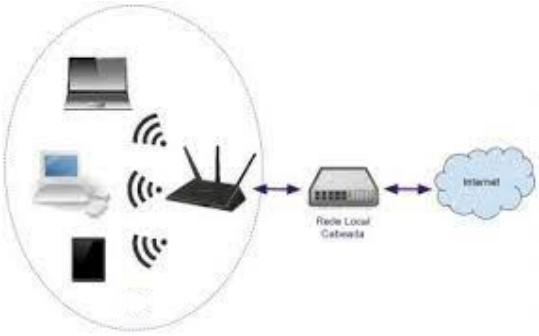

Figura 1: Topologia da rede.

O software do cliente é um programa de terminal, sem interface gráfica, em que o usuário digita os comandos e mensagens a serem enviados e recebe as respostas. São utilizadas as cores verde, amarela e vermelho no texto para indicar sucesso, informação ou erro, respectivamente. O cliente possui um sistema de autocomplete dos comandos, mostrando uma lista com os comandos possíveis enquanto o usuário digita, o usuário pode utilizar as setas para selecionar o comando desejado. Além disso, é possível utilizar comandos anteriormente digitados utilizando as setas para cima e para baixo.

Na camada de transporte, foi utilizado o protocolo TCP, por ser um protocolo que possui recursos que garantem a confiabilidade da entrega dos pacotes, não sendo necessário que esses sejam implementados na aplicação, economizando tempo e esforço. A garantia de entrega é essencial nessa aplicação por ser um sistema de bata-papo, sendo o tempo de entrega um requisito secundário.

B. Análise de Resultados

Como resultado das escolhas feitas e dos requisitos utilizados, foi desenvolvido um cliente e um servidor para obter o bate-papo desejado. Para isso, cria-se um arquivo .py e importa o socket. Em seguida, instancia o socket e utiliza umas funções do socket para o servidor poder ”ouvir”o cliente que solicita a conexão. A figura 14 mostra a conexão servidor-cliente e a figura 23 mostra a conexão cliente-servidor (linha 35).

Após o cliente está conectado ao servidor, ele poderá enviar informações ao servidor que tomará alguma ação quando essa informação chegar. Para isso, é utilizado o método send para enviar e o método recv para receber. Para fazer isso, foi criado uma função que trata essa comunicação tanto no cliente quanto no servidor, como mostrado nas figuras 22 (linha 80 a 89) e 18, respectivamente.

Para o servidor compreender as informações recebidas pelo cliente, cria-se a definição dos comandos (figura 13), verifica se a informação está dentre os comandos definidos (figura 20), se sim, chama a função que executa o comando (figura 19) e retorna uma resposta ao cliente que pode ser um erro, confirmação ou o resultado da consulta.

Os comandos foram definidos pelo protocolo IRC, mas nem todos os comandos do protocolo foram utilizados, apenas aqueles propostos no roteiro do projeto. Com isso, os comandos implementados são explicados sobre seu funcionamento nas subseções a seguir. Vale ressaltar que, em todos os comandos que possuem parâmetros mínimos, caso o cliente não envie todos os parâmetros, o servidor envia uma mensagem de erro ao cliente.

1) USER: registra username, hostname, realname, servername do cliente junto ao seu socket. A implementação feita é mostrada na figura 11.

2) NICK: registra apelido do cliente junto ao seu socket, além de verificar se o apelido escolhido jã está sendo utilizado por outro cliente, se sim, retorna uma mensagem de erro ao cliente. Foi implementado como mostrado na figura 7.

3) JOIN: entra no canal enviado junto ao comando pelo cliente (parâmetro). O canal pode ou não existir, se não existir, o canal é criado. O código desenvolvido é mostrado na figura 5.

4) LIST: não possui parâmetro e lista todos os canais que possuem clientes, ou seja, canais que existem no mesmo que o cliente fez a requisição. A implementação feita é mostrada na figura 6.

5) PART: sai do canal que o cliente está participando. Se o cliente enviar como parâmetro um canal que difira ao que ele participa ou não existe, uma mensagem de erro é enviada a ele. Foi implementado como mostrado na figura 8.

6) QUIT: fecha a conexão entre cliente e servidor. Não possui parâmetros e o desenvolvimento é mostrado na figura 10.

7) WHO: mostra informação acerca do cliente passado pelo parâmetro ou acerca dos membros participantes do canal passado pelo parâmetro. Pode enviar mais de um parâmetro. A implementação feita é mostrada na figura 12.

8) PRIVMSG: envia mensagem privada a um usuário ou aos membros do canal passado pelo parâmetro. Foi implementado como mostrado na figura 9.

Algumas funções auxiliares foram desenvolvidas para melhor visibilidade do código, dentre elas estão as funções para buscar informação do cliente a partir do socket: buscar apelido (figura 15), buscar cliente (figura 17) e buscar canal (figura 16).

Para analisar a troca de informações entre o cliente e o servidor, foi utilizado o programa Wireshark. Ao selecionar a interface de rede que deseja-se analisar, é mostrado todos os dados sendo transmitidos nela. É possível utilizar vários filtros para visualizar mais claramente o que se deseja. Ao filtrar por ”irc”, é mostrado as transmissões de dados que seguem o protocolo IRC. A figura 2 mostra as trocas de mensagens entre cliente e servidor ao se utilizar os comandos /USER e /NICK. O retângulo identificado pelo número 1 mostra que é uma aplicação IRC. O retângulo identificado pelo número 2 mostra o endereço IP da fonte e do destino (cliente e servidor). O retângulo identificado pelo número 3 mostra o cabeçalho da camada de rede.

A figura 3 mostra os dados da camada de transporte. O retângulo identificado pelo número 1 mostra que o protocolo utilizado é o TCP. O retângulo identificado pelo número 2 mostra as portas do cliente e do servidor. O retângulo identificado pelo número 3 mostra o cabeçalho da camada de transporte.

A figura 4 mostra a carga útil transmitida. É possível ver que o dado transmitido é o comando ”/NICK rafa”, que é o esperado.

IV. Vídeo de demostração

Foi gravado um vídeo mostrando o funcionamento da aplicação, sua implementação e a análise de pacotes com o Wireshark. O vídeo está disponível no endereço https://drive.google.com/file/d/1L2ES-LOUvRznYi2gfPi129KVkPDaNIzw/view?usp=share$_{l}ink$.

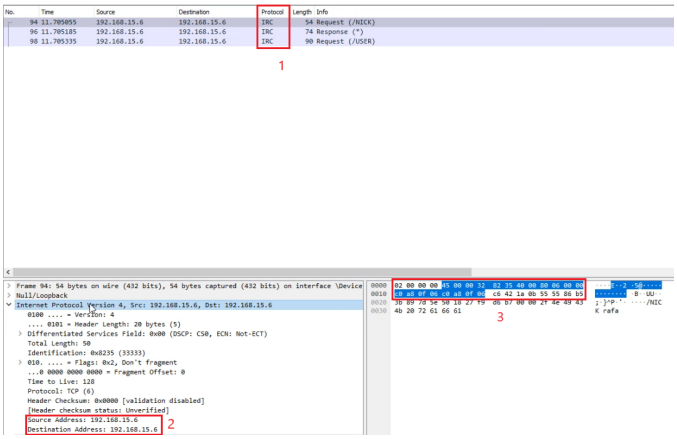

Figura 2: Análise da camada de rede no Wireshark.

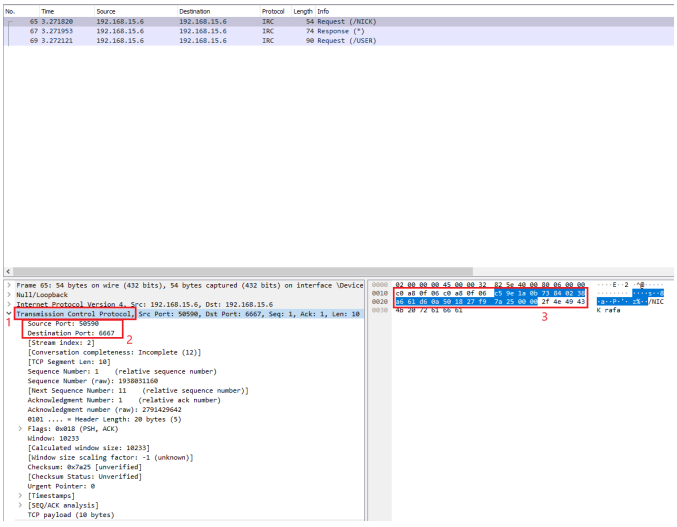

Figura 3: Análise da camada de transporte no Wireshark.

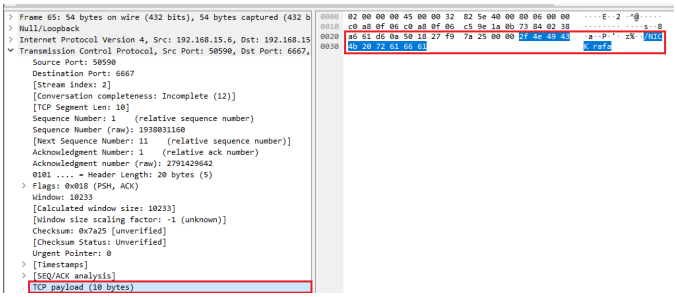

Figura 4: Análise da carga útil no Wireshark.

```python
def handle_join(*args):
    client_socket, params = args
    channel = params[0]
    old_channel = find_channel_by_client_socket(client_socket)

    if old_channel and old_channel != channel:
        remove_member_from_channel(client_socket, old_channel)

    if channel not in channels:
        channels[channel] = [client_socket]
        return f"* Você agora está no canal {channel}"
    elif client_socket not in channels[channel]:
        client = find_client_by_socket(client_socket)
        send_message_to_channel(channel, f"* {client['nick']} entrou")
        channels[channel].append(client_socket)
        return f"* Você agora está no canal {channel}"
    else:
        return f"* Você já está no canal {channel}"
```

Figura 5: Servidor - Implementação do comando join.

```python
def handle_list(*args):
    response = "*** Channel\tUsers"
    for channel, members in channels.items():
        response += f"\n*** {channel}\t{len(members)}"
    return response
```

Figura 6: Servidor - Implementação do comando list.

```python
def handle_nick(*args):
    client_socket, params = args
    newnick = params[0]
    client = find_client_by_socket(client_socket)

    if newnick in [client["nick"] for client in clients if "nick" in client]:
        return f"ERR_NICKNAMEINUSE"

    if "nick" not in client:
        client["nick"] = newnick
        return f"* Nickname atribuido para {newnick}"
    else:
        oldnick = client["nick"]
        client["nick"] = newnick
        channel = find_channel_by_client_socket(client_socket)
        if channel:
            send_message_to_channel(
                channel, f"* {oldnick} alterado para {newnick}", client_socket)
        return f"* Nickname alterado para {newnick}"
```

Figura 7: Servidor - Implementação do comando nick.

```python
def handle_part(*args):
    client_socket, params = args
    channel = find_channel_by_client_socket(client_socket)
    if not channel:
        return f'ERR_NOTONCHANNEL'

    if channel in params:
        remove_member_from_channel(client_socket, channel)
        return f"* Você saiu do canal {channel}"
    else:
        return f'ERR_NOTONCHANNEL'
```

Figura 8: Servidor - Implementação do comando part.

V. Conclusões

A aplicação desenvolvida utiliza os conceitos de protocolo TCP e IRC, além de aplicá-los na prática por meio de uma aplicação de bate-papo em Python. E os resultados do wireshark comprovam que esse objetivo foi concluído. Todos os comandos propostos pelo roteiro do projeto relacionado ao protocolo IRC foram implementadosm, testados e demonstrados por meio do vídeo explicativo.

```python
def handle_privmsg(*args):
    client_socket, params = args
    params = ' '.join(params)
    names, message = params.split(":", 1)
    names = names.strip().split(",")

    for name in names:
        if name in [client["nick"] for client in clients]:
            send_message_to_client(name, message)
        elif name in channels:
            send_message_to_channel(name, message, client_socket)
```

Figura 9: Servidor - Implementação do comando privmsg.

```python
def handle_quit(*args):
    client_socket = args[0]
    remove_client_from_server(client_socket)
    client_socket.close()
    sys.exit()
```

Figura 10: Servidor - Implementação do comando quit.

```python
def handle_user(*args):
    client_socket, params = args
    params = ' '.join(params)
    user_infos, realname = params.split(":", 1)
    user_infos = user_infos.split(" ")

    client = find_client_by_socket(client_socket)
    client["username"] = user_infos[0]
    client["hostname"] = user_infos[1]
    client["servername"] = user_infos[2]
    client["realname"] = realname
```

Figura 11: Servidor - Implementação do comando user.

```python
def handle_who(*args):
    client_socket, params = args
    names = ' '.join(params).split(',')

    for name in names:
        if name in channels.keys():
            response = f'\n*** Usuários no canal {name}'
            response += f'\n*** Username\tNickname\tHostname\tServername\trealname'
            for user in channels[name]:
                client = find_client_by_socket(user)
                response += f'\n*** {client["username"]}\t{client["nick"]}\t{client["hostname"]}\t{client["servername"]}\t{client["realname"]}'
                client_socket.send(response.encode("utf-8"))
        else:
            who_clients = [
                client for client in clients if client["nick"] == name]
            if len(who_clients) > 0:
                client = who_clients[0]
                response = f'\n*** Informações do usuário {name}'
                response += f'\n*** Username\tNickname\tHostname\tServername\trealname'
                response += f'\n*** {client["username"]}\t{client["nick"]}\t{client["hostname"]}\t{client["servername"]}\t{client["realname"]}'
                client_socket.send(response.encode("utf-8"))
```

Figura 12: Servidor - Implementação do comando who.

Referências

[1] CMU-RFC 1459-03. Disponível em: http://www.cs.cmu.edu/ srini/15-441/F06/project1/rfc.html. Acesso em: 19 jan. 2023.

[2] Kurose, J. and Ross, K. (2012) Computer Networking: A Top-Down Approach. Pearson, 6th Edition.

[3] Protocolo TCP/IP: o que é e como funciona. Disponível em: https://www.hostgator.com.br/blog/o-que-e-protocolo-tcp-ip/.


```python
COMMANDS = {
    "/JOIN": {
        "min_params": 1,
    },
    "/QUIT": {
        "min_params": 0,
    },
    "/USER": {
        "min_params": 4,
    },
    "/NICK": {
        "min_params": 1,
    },
    "/LIST": {
        "min_params": 0,
    },
    "/PART": {
        "min_params": 1,
    },
    "/WHO": {
        "min_params": 1,
    },
    "/PRIVMSG": {
        "min_params": 2,
    }
}
```

Figura 13: Servidor - Declaração dos comandos e a quantidade miníma de parâmetros.

```python
socket_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ip = get_ip()
port = 4400
socket_server.bind((ip, port))

clients = []
channels = {}

socket_server.listen()
print(f"Servidor iniciado em {ip} na porta {port}")
```

Figura 14: Servidor - Implementação da conexão entre servidor e cliente.

```python
def find_socket_by_nickname(nick):
    for client in clients:
        if client["nick"] == nick:
            return client["socket"]
```

Figura 15: Servidor - Função para encontrar apelido do usuário.

```python
def find_channel_by_client_socket(client_socket):
    for channel, members in channels.items():
        for member in members:
            if member == client_socket:
                return channel
```

Figura 16: Servidor - Função para encontrar canal.

```python
def find_client_by_socket(client_socket):
    for client in clients:
        if client["socket"] == client_socket:
            return client
```

Figura 17: Servidor - Função para encontrar cliente.

```python
while True:
    try:
        client_socket, client_address = socket_server.accept()
        print("Nova conexão a partir de", client_address)
        client_thread = threading.Thread(
            target=handle_client, args=(client_socket,))
        client_thread.start()
    except KeyboardInterrupt:
        print("\nEncerrando servidor...")
        os._exit(1)
```

Figura 18: Servidor - Implementação de um loop na comunicação servidor-cliente.

```python
handlers = {
    "/JOIN": handle_join,
    "/QUIT": handle_quit,
    "/USER": handle_user,
    "/NICK": handle_nick,
    "/LIST": handle_list,
    "/PART": handle_part,
    "/WHO": handle_who,
    "/PRIVMSG": handle_privmsg
}


def process_command(command, params, client_socket):
    error = validate_command(command, params)
    if error is not None:
        return error

    if command not in ["/USER", "/NICK", "/QUIT"]:
        registered = verify_registration(client_socket)
        if not registered:
            return f'ERR_NOTREGISTERED'

    return handlers[command](client_socket, params)
```

Figura 19: Servidor - Função para diferenciar comando de mensagem.

```python
def validate_command(command, params):
    if command not in COMMANDS:
        return "ERR_UNKNOWNCOMMAND"

    min_params = COMMANDS[command]["min_params"]

    if len(params) < min_params:
        return f"ERR_NEEDMOREPARAMS"
```

Figura 20: Servidor - Função para validar o comando enviado pelo cliente.

```python
def verify_registration(client_socket):
    client = find_client_by_socket(client_socket)
    if (not "nick" in client) or (not "username" in client):
        return False
    return True
```

Figura 21: Servidor - Função para verificar se o cliente foi cadastrado.

```python
def handle_output():
    while True:
        data = client_socket.recv(1024).decode("utf-8")
        if data:
            if data.startswith("ERR"):
                print(colored(data, "red"))
            elif data.startswith("*"):
                print(colored(data, "yellow"))
            else:
                print(data)
        else:
            os._exit(1)


output_thread = threading.Thread(target=handle_output)
output_thread.start()

commands_completer = WordCompleter(
    ['/JOIN', '/LIST', '/NICK', '/PART', '/QUIT', '/USER', '/PRIVMSG', '/WHO'])

session = PromptSession()

while True:
    with patch_stdout(raw=True):
        try:
            data = session.prompt('> ', completer=FuzzyCompleter(
                commands_completer), auto_suggest=AutoSuggestFromHistory())
            client_socket.send(data.encode("utf-8"))
        except KeyboardInterrupt or EOFError:
            print("Saindo...")
            client_socket.send("/QUIT".encode("utf-8"))
            break
```

Figura 22: Cliente - Configuração para melhorar interface da aplicação.

```python
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

clear_terminal()

while True:
    global server_ip
    global server_port
    server_ip = input("Digite o IP do servidor IRC: ")
    server_port = input("Digite a porta do servidor IRC: ")
    try:
        client_socket.connect((server_ip, int(server_port)))
        clear_terminal()
        print(
            colored(f"Conectado ao servidor {server_ip} na porta {server_port}", "green"))
        break
    except:
        print(colored("Não foi possível conectar ao servidor. Tente novamente.", "red"))

while True:
    nickname = input("Escolha um apelido: ")
    response = client_socket.send(f"/NICK {nickname}".encode("utf-8"))
    if response == 'ERR_NICKNAMEINUSE':
        print("Este apelido já está em uso. Escolha outro.")
    else:
        username = nickname
        hostname = socket.gethostname()
        servername = server_ip
        realname = getpass.getuser()
        client_socket.send(
            f"/USER {username} {hostname} {servername} :{realname}".encode("utf-8"))
        break
```

Figura 23: Cliente - Comunicação entre cliente-servidor.