Universidade de Brasília — Faculdade UnB Gama

Disciplina: FGA 0211 - Fundamentos de Redes de Computadores

Curso: Engenharia de Software **Semestre:** 2022.1

Turma: T01 **Professor:** Fernando William

Trabalho final: Criando ambientes virtuais de conversação com uso system call select()

Salas de bate-papo virtuais

Feito em Grupo

Repositório:

https://github.com/Victor-Buendia/trabfinalFRC

ALUNO	MATRÍCULA
Victor Buendia Cruz de Alvim	190020601
Yan Andrade de Sena	180145363
João Vitor de Souza Durso	180123459

Objetivos do Projeto de Pesquisa

Este trabalho final possui o intuito de permitir que o aluno compreenda a arquitetura de aplicações de rede (segundo na arquitetura TCP/IP) que envolvam gerência de diálogo. Dessa forma, os alunos irão construir uma aplicação que disponibiliza salas de bate-papo virtuais, nas quais os clientes possam ingressar e interagir. Além disso, os alunos deverão se atentar a utilizar a system call select(), que será explicada nos tópicos a seguir.

Descrição do Problema

A criação de salas virtuais possui como principal problema saber gerenciar uma comunicação full-duplex. Em uma comunicação full-duplex, ambas as partes comunicantes podem transmitir e receber mensagens simultaneamente.

Ademais, outro obstáculo importante para a criação de salas virtuais é entender a multiplexação de E/S. Sugere-se o uso da multiplexação de E/S quando o cliente está manipulando vários descritores, quando o cliente manipula vários sockets simultaneamente, quando o TCP manipula *listening* soquetes e outros soquetes conectados, quando o servidor lida com UDP e TCP em simultâneo, e/ou quando o servidor manipula vários protocolos e serviços ao mesmo tempo.

Para o desenvolvimento da aplicação deste trabalho, é necessário que, quando as condições de E/S estiverem válidas, o kernel deve ser avisado e ele notifique o usuário. Para isso, utiliza-se o system call select(). O select() é uma função que permite que um programa monitore vários descritores de arquivo, esperando até que um ou mais descritores de arquivo fiquem “prontos” para alguma classe de operação. Isto é, a função select() instrui o kernel a “acordar” o processo quando um conjunto de eventos ocorrer ou quando um certo intervalo de tempo tiver ocorrido. Um arquivo descritor é considerado pronto se for possível realizar uma operação de I/O correspondente (por exemplo, read(2), ou um small write(2)) sem bloqueio.

Por fim, a aplicação também deverá ser responsável em conseguir:

- Criar salas virtuais com nome da sala e limite de participantes;
- Listar participantes de uma determinada sala;
- Permitir ingresso de clientes, com um identificador, em uma sala existente, conforme o limite admitido para a sala;
- Saída de clientes de uma sala em que estava participando;
- Diálogo entre os clientes das salas.

Metodologia Utilizada

Para solucionar os problemas propostos, o nosso grupo se reuniu e decidiu que cada membro deveria entender as dependências do projeto, como, por exemplo, a utilização da função select(), através da consulta de sites e/ou cursos. Essa decisão foi importante para trabalhar uma cooperação dos membros do grupo, em que um apoiaria o outro em sua própria jornada para configurar a rede proposta, compartilhando conhecimento.

Os membros João Vitor e Victor Buendia se reuniram na mesma casa durante o final de semana para um pareamento mais intenso e colaboração para a construção mais eficiente da aplicação proposta. O membro Yan apoiou a construção de outros materiais remotamente.

Assim posto, João Vitor e Victor Buendia explicaram a solução que havia sido concebida para Yan, com o intuito de compreender as competências a serem desenvolvidas nesta atividade.

Solução do Problema

Para desenvolver a aplicação, ir-se-á utilizar a linguagem de programação python. Dessa forma, criou-se dois arquivos, uma para o cliente e outro para o servidor. Para rodar o projeto é preciso compilar primeiro o arquivo **chat_server.py,** utilizando o comando "*python3.10 chat_server.py"*.

Quando compilado o arquivo, as seguintes opções serão apresentadas via terminal:

```
================== MENU ==================
/CRIAR -----> Cria uma nova sala.
/LISTAR -----> Listar salas criadas.
/PARTICIPANTES -----> Listar participantes de uma determinada sala.
/SAIR -----> Encerra a aplicação.
/LOGS -----> Exibir logs.
Digite um comando:
```

Imagem 1: Menu para seleção de ações

Para criar uma sala, deve-se digitar /CRIAR (o programa aceita as variações de letra maiúscula e minúscula). Em seguida, será solicitado um nome para a sala e o seu limite máximo de participantes. Ademais, executa-se no terminal cliente, o arquivo chamado chat_client.py, utilizando o comando "*python3.10 chat_client.py*".

Quando o arquivo for executado, pedirá para o cliente inserir um *username*:

```
yan@frc:~/Área de Trabalho/frc/trabfinalFRC$ python3.10 chat_client.py
Username:
```

Imagem 2: Compilação de chat_client.py

Com o *username* inserido, o programa irá solicitar ao cliente em qual sala ele quer se conectar. Caso o cliente informe um nome de sala de forma equivocado, a aplicação irá retornar um erro e aguardar até a inserção de um nome de sala existente.

```
Username: Yan
Escolha a sala: Teste
Para sair, digite o comando /SAIR.

Yan > *** ENTROU NA SALA ***

Yan >
```

Imagem 3: Inserindo um nome de usuário e um nome de sala existente

Dentro de uma mesma sala válida, a aplicação ficará disponível para os clientes enviarem mensagens entre si. Para o cliente sair da sala e trocar para uma outra, basta digitar “/SAIR”.

A seguir há um diálogo entre dois participantes em uma sala virtual de nome “Teste”:

```
Escolha a sala: Teste
Para sair, digite o comando /SAIR.

Yan > *** ENTROU NA SALA ***

Buendia > *** ENTROU NA SALA ***

Yan > Olá

Buendia > Olá

Yan >
```

Imagem 4: Sala Teste com Yan e Buendia conversando na sala (chat_client.py)

Utilizando o comando “/LISTAR” no terminal servidor, é possível ver as salas abertas no momento, com a quantidade de usuários conectados e o limite de conexões da sala.

```
****** SALAS ******
1: Teste - 2/2 usuários conectados.
Pressione uma tecla para continuar...
```

Imagem 5: Exibindo sala com nome e limite de usuários (chat_server.py)

```
Username: Buendia
Escolha a sala: Teste
Para sair, digite o comando /SAIR.

Buendia > *** ENTROU NA SALA ***

Yan > Olá
```

```
Buendia > Olá

Buendia >
```

Imagem 6: O cliente Buendia envia a mensagem (chat_client.py)

```
Escolha a sala: Teste
Para sair, digite o comando /SAIR.

Buendia > *** ENTROU NA SALA ***

Yan > Olá

Buendia > Olá

Buendia > *** SAIU DA SALA ***

Yan >
```

Imagem 7: Buendia sai da aplicação (chat_client.py)

```
****** PARTICIPANTES DA SALA "Teste" ******

1 - Yan
Pressione uma tecla para continuar...
```

Imagem 8: Opção de listar participantes de uma sala (chat_server.py)

Ademais, o código da aplicação será explicado abaixo, tanto o chat_server.py quanto o chat_client.py.

Chat_server.py

Nas primeiras linhas do código, são definidas as informações básicas do servidor, como o tamanho do HEADER, o IP, e a PORT:

```
HEADER_LENGTH = 10

IP = "127.0.0.1"
PORT = 1234
```

Depois, defini-se as opções do menu:

```
comandos = {
    "/CRIAR": "Cria uma nova sala.",
    "/LISTAR": "Listar salas criadas.",
    "/PARTICIPANTES": "Listar participantes de uma determinada sala.",
    "/ENCERRAR": "Encerrar uma sala.",
    "/SAIR": "Encerrar a aplicação.",
    "/LOGS": "Exibir logs"
}
```

Declara-se também um vetor de logs que irá armazenar os logs da aplicação:

```
logs = [ ]
```

Criou-se um socket com conexão TCP, a partir do código:

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

A seguir, realizou-se um bind, para que o servidor informe ao sistema operacional que ele usará determinado IP e PORT já configurados:

```
server_socket.bind((IP, PORT))
```

O servidor agora estará ouvindo novas conexões:

```
server_socket.listen()
```

Define-se uma lista de sockets para o select():

```
sockets_list = [server_socket]
```

Depois, declara-se um dicionário de clientes conectados (socket como chave, header do usuário e nome) :

```
clients = { }
```

Define-se também a listas de salas abertas. contendo limite de usuários e usuários conectados:

```
rooms = [ ]
```

Agora, tratar-se-á de quando o usuário é adicionado a uma sala:

```
def add_user_to_room(room_name, user):
```

A sala selecionada inicializa com valor ‘None’, em seguida, faz-se um laço dentro do vetor **rooms**, onde todas as salas estão armazenadas:

```
room_selected = None
   for room in rooms:
```

A seguir, é feita a verificação do nome da sala inserida pelo cliente, se ela existir a sala selecionada recebe as informações da sala:

```
if room['name'] == room_name:
          room_selected = room
```

Se a sala não existir, o cliente é notificado para inserir uma sala com nome existente.

```
else:
      log_message(f'{user["username"]} tentou entrar em sala inexistente.')
      data = {'username': 'Sistema', 'message': '404'}
      encoded_data = json.dumps(data).encode('utf-8')
      user['socket'].send(encoded_data)

      user['socket'].shutdown(socket.SHUT_RDWR)
      user['socket'].close()
```

Se a sala selecionada for diferente de ‘None’, então o limite da sala escolhido é definido:

```
if room_selected is not None:
      room_limit = room_selected['limit']
```

Agora, verifica-se um usuário pode ingressar em uma sala, se ela estiver cheia o cliente é notificado e não consegue entrar em uma sala. Se não, o cliente consegue entrar na sala com êxito.

```
if (len(room_selected['users']) + 1) <= room_limit:
```

```
            room_selected['users'].append(user)
            user['room'] = room_selected
            clients[user['socket']] = user

            notify_user_join_room(
                username_join=user['username'], room=room_selected)
            log_message(
                f'{user["username"]} entrou na sala {room_selected["name"]}.')

            return room_selected
        else:
            log_message(
                f'{user["username"]} tentou entrar na sala \'{room_selected["name"]}\', mas seu
limite de participantes já foi atingido.')
            data = {'username': 'Sistema', 'message': '501'}
            encoded_data = json.dumps(data).encode('utf-8')
            user['socket'].send(encoded_data)

            user['socket'].shutdown(socket.SHUT_RDWR)
            user['socket'].close()
```

Sobre a criação de salas, primeiro é solicitado o nome da sala e o limite dela para ele. Em seguida, gera-se um id e os dados são salvos. Além disso, é feita uma validação desses dados e, se os dados estiverem corretos, a sala é criada, se não, espera o usuário inserir dados corretos.

```
def create_room():
    print('\nCriando sala...')

    try:
        room_name = input('Digite o nome da sala: ')
        room_limit = input('Digite o limite da sala: ')
        room_id = (rooms[len(rooms) - 1]['id'] + 1) if (len(rooms) > 0) else 1
        rooms.append({'id': room_id, 'name': room_name,
                      'limit': int(room_limit), 'users': [ ]})
    except ValueError as e:
        print('\nInsira dados válidos.')
        time.sleep(3)
        clear()
        create_room()

    clear()
    print(f'Sala {room_name} criada.')
    log_message(
        f'Sala {room_name} criada. Atualmente, existem {len(rooms)} salas criadas.')
```

Abaixo está a parte de excluir salas virtuais. Primeiro, as salas disponíveis são exibidas com seu ID, depois o usuário deve inserir o ID da sala a ser removida. Em seguida, é feito um laço procurando a sala selecionada, caso seja encontrada, ela é removida. Caso não, notifica-se o usuário.

```
def delete_room():
    show_rooms()
    room_to_be_removed = None

    id_room_to_be_removed = input('Selecione o id da sala a ser removida:')
    for room in rooms:
        if room['id'] == int(id_room_to_be_removed):
            room_to_be_removed = room
            continue

    if room_to_be_removed is not None:
        remove_users_from_room(room=room_to_be_removed)
        rooms.remove(room_to_be_removed)
        print("Sala encerrada")
        log_message(f'Sala {room_to_be_removed["name"]} foi encerrada.')
        threading.Thread(target=manage_sockets).start()

    else:
        print('Sala não encontrada.')
```

Agora é aprensentada a inicialização do programa. Ele dá início a uma thread que possui o select() para fazer o recebimento de informações dos descritores vindos dos clientes, e em seguida exibe o menu que antes foi declarado no começo do código. Esse menu fica executando um loop infinito validando se o que o usuário digitar é uma opção válida ou não.

```
def init():
    threading.Thread(target=manage_sockets).start()
    log_message(f'Esperando conexões em {IP}:{PORT}...')
    option = ''
    first_time = 0
    while True:
        print(option)
        match option.upper():
            case '/CRIAR':
                create_room()

            case '/LISTAR':
                show_rooms()
                input('Pressione uma tecla para continuar...')
                print('\n\n')

            case '/LOGS':
```

```
            show_logs()

        case '/ENCERRAR':
            delete_room()

        case '/PARTICIPANTES':
            show_users_in_room()
            input('Pressione uma tecla para continuar...')
            print('\n\n')

        case '/SAIR':
            print('\nEncerrando aplicação...')
            os._exit(1)

        case _:
            if first_time != 0:
                print('\nComando Inválido!')
                time.sleep(1)
            first_time = 1
    clear()
    option = menu()
```

No final da função de inicialização também é impressa a barra de menu, onde irá receber os comandos do usuário:

```
def menu():
    print('================== MENU ==================')
    for comando in comandos:
        print(f'{comando} -----> {comandos[comando]}')
    try:
        option = input('Digite um comando: ')
    except KeyboardInterrupt as e:
        print('\n\nO programa foi finalizado abruptamente.')
        os._exit(1)

    return option
```

Este projeto também possui uma função que salva a mensagem dos logs, a qual recebe uma mensagem e salva a mesma.

```
def log_message(message):
    logs.append(message)
```

Agora tratar-se-á do recebimento das mensagens vindo de chat_client.py:

```
def receive_message(client_socket):
   try:
```

Recebe-se o header com recv contendo o tamanho da mensagem:

```
message_header = client_socket.recv(HEADER_LENGTH)
```

Se os dados não forem recebidos, o servidor irá encerrar a conexão, usando socket.close() ou socket.shutdown(socket.SHUT_RDWR):

```
if not len(message_header):
          return False
```

Converte-se o header da mensagem para inteiro:

```
message_length = int(message_header.decode('utf-8').strip())
```

Retorna-se o header da mensagem e os dados recuperados com o recv também:

```
return {'header': message_header, 'data': client_socket.recv(message_length)}
```

Caso o cliente tenha encerrado a conexão abruptamente ou perdeu a conexão, o programa retorna falso:

```
  except:
      return False
```

Em seguida, é feito um laço, onde os sockets válidos presentes dentro da lista de sockets são checados. Lê-se as informações dos sockets presentes e realiza-se a operação select() para esperar o recebimento de mensagens dos servidores:

```
def manage_sockets():
   while True:
      check_sockets_list()
      read_sockets, _, exception_sockets = select.select(
         sockets_list, [ ], sockets_list)
```

Aqui na função de verificar se o socket está presente na lista de soquetes, se ele não estiver mais ativo, ele é excluído da lista.

```
def check_sockets_list():
    for checking_socket in sockets_list:
        if checking_socket.fileno() == -1:
            sockets_list.remove(checking_socket)
```

É feita uma iteração pelos sockets de leitura para recebimento de mensagem:

```
    for notified_socket in read_sockets:
```

Se o socket notificado é um socket servidor, a conexão é validada.

```
        if notified_socket == server_socket:
```

Aceitando uma nova conexão, dá-se um novo socket (socket agora do cliente), conectado apenas a esse determinado cliente. O outro objeto retornado é ip/port set.

```
            client_socket, client_address = server_socket.accept()
```

Recebe-se o nome do cliente:

```
            user = receive_message(client_socket)
```

Se o nome do usuário for falso significa que o mesmo caiu antes de mandar o nome:

```
            if user is False:
                continue
```

Adiciona-se o socket do cliente à lista de sockets olhados pelo select().

```
            sockets_list.append(client_socket)
```

Salva-se o nome de usuário e o header e também codificamos essas informações num objeto JSON:

```
            data = json.loads(user['data'].decode('utf-8'))
             username = data['username']
             room = data['room']
```

O usuário é adicionado a uma sala (escolhida pelo cliente):

```
            add_user_to_room(room_name=room, user={
                        'socket': client_socket, 'username': username})
```

Se o socket já está na lista, então o mesmo está apenas enviando uma mensagem. A mensagem é então recebida para distribuí-la aos clientes.

```
        else:
            send_message(socket=notified_socket)
```

Agora, verifica-se se um socket foi notificado dentro de uma lista dos sockets que receberam algum tipo de exceção (falha). Tenta-se remover esse socket e se não for possível excluí-lo, uma exceção é lançada.

```
    for notified_socket in exception_sockets:
        try:
            sockets_list.remove(notified_socket)
            del clients[notified_socket]
        except Exception as e:
            log_message(f'Erro ao remover socket que deu errado: {e}')
```

A função de notificar quando um usuário sai da sala é simples. Realiza-se um laço *for* para enviar os dados de nome de usuário e a mensagem que ele saiu da sala para todos os clientes conectados na sala em que o usuário saiu. Essa mensagem é codificada em JSON e em seguida tenta-se enviar a mensagem. Caso dê errado, uma exceção é lançada:

```
def notify_user_left_room(username_left, room):
    for user in room['users']:
        data = {'username': username_left, 'message': '*** SAIU DA SALA ***'}
        encoded_data = json.dumps(data).encode('utf-8')
        try:
            user['socket'].send(encoded_data)
        except Exception as e:
            log_message(f'Não foi possível enviar mensagem: {e}')
```

A função de notificar quando um usuário entra na sala funciona da seguinte maneira: um laço *for* é realizado para enviar os dados de nome de usuário e a mensagem que ele entrou na sala para cada um dos clientes já conectados naquela sala. A informação é codificada em formato JSON e, em seguida, enviada.

```
def notify_user_join_room(username_join, room):
    for user in room['users']:
```

```
        data = {'username': username_join, 'message': '*** ENTROU NA SALA ***'}
        encoded_data = json.dumps(data).encode('utf-8')
        user['socket'].send(encoded_data)
```

A função de remover usuários de uma sala executa um laço *for* para enviar uma mensagem de sala encerrada e encerrar a conexão do socket para todos os clientes que estavam na sala.

```
def remove_users_from_room(room):
    for user in room['users']:
        data = {'username': 'Sistema',
                'message': '*** ESTA SALA FOI ENCERRADA ***'}
        encoded_data = json.dumps(data).encode('utf-8')
        user['socket'].send(encoded_data)

        data = {'username': 'Sistema',
                'message': '201'}
        encoded_data = json.dumps(data).encode('utf-8')
        user['socket'].send(encoded_data)

        user['socket'].shutdown(socket.SHUT_RDWR)
        user['socket'].close()
```

Ademais, a função de remoção dos sockets funciona da seguinte maneira: os dados do usuário que saiu são coletados, na sala que o usuário saiu e em seguida, remove-se o usuário da lista de usuários. Uma notificação é enviada na sala que o usuário saiu e em seguida o seu socket é excluído. Em seguida, uma mensagem de log é emitida, afirmando que a conexão do usuário foi encerrada.

```
def remove_socket(socket_closed):
    user_left = clients[socket_closed]
    room_user_left = user_left['room']
    room_user_left['users'].remove(user_left)

    notify_user_left_room(
        username_left=user_left['username'], room=room_user_left)

    del clients[socket_closed]
    sockets_list.remove(socket_closed)
    log_message(f'Conexão encerrada de: {user_left["username"]}')
```

A função de envio de mensagem começa recebendo uma mensagem do socket cliente e verificando se essa mensagem é falsa (se a mensagem for falsa, então o cliente desconectou, então podemos removê-lo da lista de sockets). Se a mensagem não for *False*, pega-se o usuário do socket notificado, para saber quem enviou a mensagem, e em seguida o log desta mensagem é gravado. Itera-se sobre os clientes conectados na sala,

distribuindo a mensagem para eles. Verifica-se se o socket é o autor, para não a mensagem não ser enviada para o próprio autor dela. A mensagem é codificada para formato JSON e então enviada.

```
def send_message(socket):
    message = receive_message(socket)

    if message is False:
        remove_socket(socket)
        return

    sender_user = clients[socket]
    username = sender_user['username']

    log_message(
        f'{username} [{sender_user["room"]["name"]}]: {message["data"].decode("utf-8")}')

    for user in sender_user['room']['users']:

        if user['socket'] != socket:
            data = {'username': username,
                    'message': message['data'].decode('utf-8')}
            encoded_data = json.dumps(data).encode('utf-8')

            user['socket'].send(encoded_data)
```

A função de mostrar o log é muito simples. Ela apenas realiza um laço for, pega as mensagens e outras informações de log e exibe para o usuário.

```
def show_logs():
    clear()
    print('****** LOGS ******')

    for log in logs:
        print(f'- {log}')

    input('Pressione uma tecla para continuar...')
    print('\n\n')
```

A função de mostrar salas também é muito simples. Ela apenas realiza um laço *for*, pega o id das salas, o nome das salas, o número de participantes da sala que estão conectados e também o número limite de participantes da sala e exibe para o usuário.

```
def show_rooms():
    clear()
    print('****** SALAS ******')
    for room in rooms:
        print(
            f'{room["id"]}: {room["name"]} - {len(room["users"])}/{room["limit"]} usuários
conectados.')
```

A função de listar participantes em uma sala inicia atribuindo o valor da variável da sala selecionada em 'None' em seguida mostra as salas disponíveis e pede para o usuário digitar o nome da sala que deseja ver os participantes. Com o nome da sala, faz-se um laço *for* para procurar se a sala inserida está entre a lista de salas existentes. Se o nome da sala for existente, o nome da sala e o nome dos participantes são apresentados. Se a sala não existir, notifica-se o usuário que a sala não existe e pede-se pro mesmo tentar inserir outro nome de uma sala existente.

```
def show_users_in_room():
    clear()
    room_selected = None
    show_rooms()
    room_name = input(
        '\nDigite o nome da sala que deseja ver os participantes: ')
    for room in rooms:
        if room['name'] == room_name:
            room_selected = room

    users_in_room = len(room["users"])

    if room_selected is not None:
        clear()
        print(f'****** PARTICIPANTES DA SALA "{room_name}"  ******')
        i = 1
        print('')
        for user in room_selected["users"]:
            print(f'{i} - {user["username"]}')
            i+=1
    else:
        print('\nSala não encontrada. Digite outra sala.')
        time.sleep(3)
        clear()
        show_users_in_room()
```

Por fim, há a execução do início do programa, onde a função init() previamente definida é chamada e a exceção de apertar Ctrl +C no terminal é tratada, para finalizar o programa elegantemente.

```
try:
    init()
except KeyboardInterrupt as e:
    print('\n\nO programa foi finalizado abruptamente.')
    os._exit(1)
```

Chat_client.py

Nas primeiras linhas do código, definem-se as informações básicas do servidor, como o tamanho do HEADER , o IP, e a PORT (que precisam ser iguais ao que definimos no servidor):

```
HEADER_LENGTH = 10

IP = "127.0.0.1"
PORT = 1234
```

Em seguida o buffer é limpado para mostrar o prompt de digitar mensagem:

```
def limparBuffer(my_username):
    print(f'\n{my_username} > ', end='')
    sys.stdout.flush()
```

Em seguida, criou-se uma função que recebe a mensagem e decodifica a mesma, fazendo também a verificação se o programa não foi encerrado de forma forçada (apertando Ctrl +C, por exempo). Se o programa não foi finalizado, a função retorna a mensagem decodificada e em formato JSON.

```
def decode_message(receiver):
    message = ' '
    has_message = True

    while has_message:
        try:
            char = receiver.recv(1).decode('utf-8')
            message += char

            if char == '}':
                has_message = False
        except KeyboardInterrupt as e:
            print('\n\nO programa foi finalizado abruptamente.')
            os._exit(1)
```

```
    return json.loads(message)
```

Na função de inicialização, pede-se para o cliente inserir um nome de sala:

```
def init():
    my_room = input("Escolha a sala: ")
```

Cria-se um socket com conexão TCP:

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

Conecta-se o socket em um ip e em uma porta:

```
client_socket.connect((IP, PORT))
```

Define-se a conexão para não-bloqueante, para que o recv() não bloqueie, só retorne algum erro:

```
client_socket.setblocking(False)
```

Prepara-se o nome de usuário e também o nome da sala para enviar em seguida. Os dados são codificados para JSON e depois para bytes. Após isso, conta-se o número de bytes e prepara-se o *header* com tamanho fixo, também codificado para bytes:

```
data = {
    'username': my_username,
    'room': my_room,
}
encoded_data = json.dumps(data).encode('utf-8')
username_header = f"{len(encoded_data):<{HEADER_LENGTH}}".encode('utf-8')
client_socket.send(username_header + encoded_data)
```

Em seguida, a opção de sair é exibida e cria-se uma thread para limpar o buffer:

```
print('Para sair, digite o comando /SAIR.')
threading.Thread(target=limparBuffer(my_username)).start()
```

Realizou-se um laço *while* que faz as operações select(), e separou-se quais *readers* estão presentes no client_socket, que são o recebimento de mensagens do servidor e a leitura stdin para o envio de mensagens pro servidor:

```
while True:
    readers, _, _ = select.select([sys.stdin, client_socket], [], [])
    for reader in readers:
        if reader is client_socket:
            try:
```

Faz-se a decodificação da mensagem recebida pelo client_socket (veio do servidor) e analisa-se o conteúdo dessa mensagem verificando se o usuário que enviou ela se chama ‘Sistema’. Caso as mensagens recebidas forem do usuário ‘Sistema’ significa que são mensagens de erros, exceções que em seguida são tratadas e exibidas. Além disso, a thread faz a limpeza do buffer.

```
                message = decode_message(receiver=client_socket)
                if message["username"] == 'Sistema':
                    match message["message"]:
                        case '404':
                            print('\n\nA sala escolhida não existe. Tente novamente.')
                            time.sleep(3)
                            clear()
                            init()
                        case '501':
                            print('\n\nO limite da sala escolhida foi atingido. Escolha outra sala.')
                            time.sleep(3)
                            clear()
                            init()
                        case '201':
                            time.sleep(3)
                            clear()
                            init()
                sys.stdout.flush()
                print('\033[A \033[A')
                print(f'\n{message["username"]} > {message["message"]}')
                threading.Thread(target=limparBuffer(my_username)).start()
```

Caso não vier dados, tenta-se novamente. Caso dê errado, executa-se novamente sys.exit().

```
            except IOError as e:
                if e.errno != errno.EAGAIN and e.errno != errno.EWOULDBLOCK:
```

```
                print('Reading error: {}'.format(str(e)))
                sys.exit()
            continue

        except Exception as e:
            print('Reading error: '.format(str(e)))
            sys.exit()
```

Se não ocorreu nenhum erro, espera-se o usuário mandar a mensagem.

```
    else:
        threading.Thread(target=limparBuffer(my_username)).start()
```

Verifica-se tanto se o usuário encerrou abruptamente a conexão, quanto se ele digitou o comando de sair da sala.

```
        try:
            message = input()
        except KeyboardInterrupt as e:
            print('\n\nO programa foi finalizado abruptamente.')
            os._exit(1)

        if message.upper() == "/SAIR":
            print('Saindo da sala...')
            client_socket.shutdown(socket.SHUT_RDWR)
            client_socket.close()
            time.sleep(1)
            clear()
            init()
```

Se a mensagem não estiver vazia, ela é enviada, porém, antes codificada para bytes juntamente ao *header*.

```
        if message:
            message = message.encode('utf-8')
            message_header = f"{len(message):<{HEADER_LENGTH}}".encode('utf-8')
            client_socket.send(message_header + message)
```

Por fim, há a execução do início do programa, onde é solicitado ao usuário sua identificação, e logo em seguida a função init() previamente definida é camada. Tratou-se, também, da exceção do Ctrl +C.

```
try:
    my_username = input("Username: ")
```

```
    init()
except KeyboardInterrupt as e:
    print('\n\nO programa foi finalizado abruptamente.')
    os._exit(1)
```

Conclusão

Como conclusão, nosso projeto atende às especificações mínimas elencadas no roteiro apresentado pelo professor. A gerência de salas e configuração concentra-se no servidor e os clientes escolhem suas salas e participam de bate-papos virtuais através da conexão que possuem estabelecidas com o servidor.

A limitação no nosso projeto se encontra na parte de ter uma autonomia 100% dentro da aplicação do cliente, uma vez que para saber os nomes das salas, é preciso consultá-las pelo servidor e essa informação não é proativamente enviada para o cliente antes dele inserir seu usuário e a sala que deseja entrar.

Ademais, não conseguimos implementar a parte bônus com OAuth2.0 tendo em vista a falta de tempo para a entrega do projeto e o cansaço do final de semestre.

Aprendizados

João Vitor de Souza Durso

A execução deste projeto desenvolveu meu conhecimento no protocolo TCP/IP, pois me incentivou a colocar na prática os ensinamentos passados em sala de aula. Por meio de atividades práticas como essa, os conceitos são relembrados e o aprendizado se torna mais sólido. Ademais, através deste trabalho, pude aprender melhor como funciona o envio de mensagens com sockets e como utilizá-los, por meio da linguagem Python.

Além disso, puder compreender o uso *system call* **select()**, a responsável por resolver o problema citado na introdução do projeto, isto é, na criação de um diálogo full-duplex. Ela faz o monitoramento do envio e recebimento via sistema operacional, multiplexando as entradas e saídas do programa. Isso é importante, porque vários clientes e o servidor devem enviar informações simultaneamente, e nesse caso, o sistema operacional não pode perder essas mensagens, e nisso a função select() foi imprescindível.

A minha participação no projeto foi na construção do código da aplicação, tanto do cliente, como do servidor. Por meio de uma reunião presencial, eu e o Victor conseguimos implementar o funcionamento do diálogo full-duplex, utilizando a função select(). Além disso, em um primeiro momento, desenvolvi a criação de salas com limite de participantes, a listagem de participantes de uma sala e a permissão do ingresso do cliente em uma sala, de acordo com o limite. Além disso, aproveitei para refatorar o código do servidor que já estava muito complexo, colocando-o dentro de funções.

Na parte do relatório e slide, fiquei responsável por fazer uma revisão sobre o que o Yan escreveu e aproveitei para retificar a parte linguística (português).

Como pontos de melhoria para o trabalho, acredito que o tempo é imprescindível para um projeto final complexo como esse, pois precisamos entender os detalhes de cada funcionalidade, e se tivéssemos mais tempo, até uma interface gráfica seria possível. Por outro lado, como ponto positivo do projeto, foi muito mais satisfatório implementar este projeto utilizando Python, pois o código fica muito mais limpo e reutilizável do que em C, por exemplo, fora a maior facilidade em compreender cada linha de código.

Minha autoavaliação, acredito que seja um 8, pois toda a parte de funcionalidade do código foi feita por mim e pelo Victor. Por meio dessa reunião presencial, ficou muito mais fácil tirar dúvidas e compartilhar ideias/interpretações sobre o problema, a fim de entender e implementar a lógica do programa. Por fim, fiz a revisão do português do relatório e acrescentei partes que eu achei relevante.

Yan Andrade de Sena

Neste projeto foi muito satisfatório e enriquecedor, tinha mexido pouquíssimas vezes com a linguagem de programação python, mas com esse projeto pude me aprofundar um pouco mais. Usando a função select() pude ver como ela torna a gerência mais fácil entre sockets e também pude ver como um servidor e um cliente se diferenciam.

Além disso, achei bacana a parte de criptografar a mensagem para ela ser enviada para um servidor e também achei muito produtivo ver como uma conexão TCP funciona.

Já teve atividades passadas que o professor passou que pude ver as diferenças entre TCP e UDP, mas sinto que nesse projeto consegui me aprofundar mais sobre o protocolo TCP/IP.

Também gostei da forma que o servidor gerencia as funções quando elas são inseridas no terminal pelo usuário, gostei bastante da apresentação dos menus.

Tive muita dificuldade em conseguir desenvolver algo no código já que consegui me reunir poucas vezes com meu grupo. Tentei fazer a autenticação com OAuth2.0, mas não consegui rodar o código de exemplo disponibilizado na documentação do OAuth2.0 com python. Mas consegui aprender os conceitos de OAuth2.0 e também como ele seria aplicado no projeto.

Tive bastante dificuldade também em conseguir desenvolver a função de criação de salas privadas e públicas, eu sabia o que fazer em teoria, mas limitações como conhecimento da linguagem me impediram de implementar essa funcionalidade.

Em resumo, gostei de entender sobre o projeto, explicar cada linha de código neste relatório e aprender mais sobre o protocolo TCP/IP além de aplicar o mesmo.

Me dou nota 7 pelo trabalho final, gostaria de ter participado mais, porém nos dias de desenvolvimento do projeto acabei não conseguindo me reunir com os outros membros da equipe. Mas apesar disso a criação dos documentos (relatório e slides) explicando o código me deram uma base de como a aplicação funciona.

Victor Buendia Cruz de Alvim

A execução deste projeto desenvolveu muito meu conhecimento no protocolo TCP/IP utilizado na camada de transporte de uma máquina. Ver como o servidor e cliente fazem o processo de comunicação através de um IP e porta específicas e se comunicam — na prática — através do envio de mensagens por meio de sockets, que facilitam a conexão entre eles, foi algo muito importante para meu aprendizado em redes. Todo o conhecimento

que advém disso, pensando no gerenciamento dos sockets dos clientes, a redistribuição das mensagens dos clientes e a divisão das salas, exigiu um conhecimento importante sobre um diálogo TCP.

Além disso, outro aprendizado importante foi entender o uso da *system call* **select()**, que foi a responsável por resolver a dificuldade da criação de um diálogo que fosse full-duplex nessa comunicação. Isso porque essa função fazia o sistema operacional monitorar as duas partes (recebimento e envio) de mensagens nos clientes e uma parte (recebimento) no servidor. Isso foi particularmente importante pro projeto na medida em que quando uma conexão era aceita, um erro era disparado porque o servidor e o cliente tentavam imediatamente fazer uma leitura de informação, sem que ela estivesse disponível. A função **select()** permitiu, então, que o sistema operacional gerenciasse esse estado de espera e leitura tanto no cliente, como no servidor, de forma a evitar o erro e ainda permitir que os clientes enviassem mensagens simultaneamente dentro de uma mesma sala.

A minha participação foi mais enfática na construção do código da aplicação, tanto do cliente, como do servidor. A implementação do select(), criação do Menu, criação de comportamento no cliente para o tratamento de exceções acionadas no servidor, listagem de salas, saída da sala para entrada em outra e encerramento da sala são exemplos de algumas das minhas contribuições.

Na parte do relatório e slide, fiquei responsável por fazer uma revisão sobre a construção do Yan, implementando as partes faltantes e realizando correções sobre erros de interpretação do código em suas descrições.

Como pontos de melhoria para o trabalho, acredito que ter alguns checkpoints do desenvolvimento do trabalho com o professor seria produtivo para esclarecer dúvidas e apoiar os alunos no desenvolvimento do projeto. Divulgar o projeto final com mais antecedência também seria uma melhora interessante para dar maior flexibilidade e autonomia pros alunos conseguirem se organizar com mais facilidade para criarem o trabalho. De pontos positivos, vale a pena mencionar a flexibilidade do uso da linguagem na criação do trabalho (C, C++, Python…) e a existência de arquivos e experimentos com TCP antes desse trabalho (como o Lab de TCP e UDP feito em sala).

Em autoavaliação, acredito que me atribuo um 9, tendo em vista que estive bem focado na criação do código e entendimento dos conceitos e requisitos propostos pelo roteiro do professor. Tive um trabalho intenso com o João Vitor para a criação do código e acho que essa colaboração facilitou o entendimento e construção do trabalho. Além disso,

ajudar o Yan a entender o que fizemos colocou à prova meus conhecimentos e minha capacidade didática para falar sobre o que fizemos.

Referências

- Getting started with event loops: the magic of select: https://www.youtube.com/watch?v=2ukHDGLr9SI&ab_channel=ThomasBallinger
- Socket Chatroom client - Creating chat application with sockets in Python: https://www.youtube.com/watch?v=ytu2yV3Gn1I&ab_channel=sentdex
- Socket Chatroom server - Creating chat application with sockets in Python: https://www.youtube.com/watch?v=CV7_stUWvBQ&ab_channel=sentdex
- select — Waiting for I/O completion: https://docs.python.org/3/library/select.html
- json — JSON encoder and decoder: https://docs.python.org/3/library/json.html