

# Relatório Projeto Final

sexta-feira, 1 de novembro de 2024

Requisitos do Chat:

RF1: O sistema terá interface gráfica

RF2: O sistema solicita a entrada de um nome para cada usuário que entrar

RF3: O sistema exibe uma mensagem que exibe para todos que tal usuário entrou no chat

RF4: O sistema exibe uma lista de todos os usuários logados no chat

RF5: O sistema permite comunicação entre todos contendo uma opção “Todos” e um chat privado apenas selecionando o nome da lista de usuário.

## 1. Utilização dos Sockets e Tratamento Broadcast

**Cliente:** cria um socket TCP para se conectar ao servidor na porta especificada. O cliente se identifica com um nome que é enviado ao servidor logo após a conexão ser estabelecida. O cliente utiliza uma thread para receber mensagens do servidor continuamente e outro loop principal para enviar mensagens de entrada do usuário.

```
# IP e PORTA do servidor ao qual queremos nos conectar
HOST = '127.0.0.1' # Endereço do servidor
PORTA = 9999 # Porta do servidor de sockets
socket_cliente = sock.socket(sock.AF_INET, sock.SOCK_STREAM)
socket_cliente.connect((HOST, PORTA))
```

```
C:\Users\Pichau\OneDrive - Grupo Marista\2 semestre\Cisco\webchat>cliente.py
----- CHAT INICIADO -----
Informe seu nome para entrar no chat: leandro canha
```

```
C:\Users\Pichau\OneDrive - Grupo Marista\2 semestre\Cisco\webchat>cliente.py
----- CHAT INICIADO -----
Informe seu nome para entrar no chat: Fabio
```

```
C:\Users\Pichau\OneDrive - Grupo Marista\2 semestre\Cisco\webchat>servidor.py
O servidor 127.0.0.1:9999 está aguardando conexões
Conexão bem sucedida com leandro canha via endereço: ('127.0.0.1', 51090)
Conexão bem sucedida com Fabio via endereço: ('127.0.0.1', 51167)
```

**Servidor:** O socket TCP é configurado para aceitar múltiplas conexões simultâneas. Cada nova conexão de cliente é tratada em uma thread dedicada para evitar bloqueios e permitir que o servidor continue aceitando novos clientes. Para cada cliente, o servidor adiciona o nome do usuário a uma lista de clientes conectados.

```
# Endereço para o servidor
HOST = '127.0.0.1'
PORTA = 9999
lista_clientes = []
sock_server = sock.socket(sock.AF_INET, sock.SOCK_STREAM)
# Tentativa de conexão do servidor
try:
    sock_server.bind((HOST, PORTA))
    sock_server.listen()
    print(f"O servidor {HOST}:{PORTA} está aguardando conexões")
    # Loop para aceitar conexões de clientes, em caso de erro mensagem e fechamento do servidor
    while True:
        try:
            sock_conn, ender = sock_server.accept()
            thread_cliente = threading.Thread(target=recebe_dados, args=[sock_conn, ender])
            thread_cliente.start()
        except Exception as e:
            print(f"Erro ao aceitar conexão: {e}")
except Exception as e:
    print(f"Erro na inicialização do servidor: {e}")
sock_server.close()
```

Broadcast: O servidor possui uma função de broadcast, onde a mensagem recebida é enviada para todos os clientes conectados, exceto o remetente. Assim, todos os usuários são notificados em tempo real sobre novas mensagens enviadas ao chat. Para evitar que o remetente receba sua própria mensagem, o servidor verifica se o socket de origem é diferente do destino.

```
30
31 # Função para enviar mensagem a todos os clientes conectados
32 def broadcast(mensagem, origem):
33     for nome, cliente in lista_clientes:
34         if cliente != origem: # Evita que o remetente receba sua própria mensagem
35             try:
36                 cliente.send(mensagem.encode())
37             except:
38                 remover(cliente, nome)
39
```

Unicast: O servidor possui também uma função unicast, onde a mensagem recebida é enviada apenas para o usuário selecionado da lista de usuários no chat. Assim, apenas o usuário selecionado recebe a mensagem do remetente.

```
1 # Função para enviar mensagem privada para um cliente específico
2 def unicast(destinatario, mensagem, remetente):
3     for nome, cliente in lista_clientes:
4         if nome == destinatario:
5             try:
6                 cliente.send(f"{remetente} (privado) >> {mensagem}".encode())
7             except:
8                 remover(cliente, nome)
9             return
10    print(f"Cliente {destinatario} não encontrado.")
11
```

**Cliente:** foi criada uma thread para o recebimento de mensagens do servidor. Essa thread chama uma função, que escuta novas mensagens do servidor e as exibe ao usuário. Essa abordagem permite que o cliente continue enviando mensagens enquanto a thread

paralela aguarda novas mensagens. Assim, o cliente não precisa esperar a resposta do servidor antes de enviar outra mensagem

```
16  # Função para receber mensagens do servidor
17  def receber_mensagens():
18      while True:
19          try:
20              mensagem = socket_cliente.recv(1024).decode()
21              if not mensagem:
22                  print("Conexão encerrada pelo servidor.")
23                  break
24              print(mensagem)
25          except:
26              print("Erro ao receber mensagem. Conexão encerrada.")
27              socket_cliente.close()
28              break
29  # Thread para receber mensagens
30  thread_receber = threading.Thread(target=receber_mensagens)
31  thread_receber.start()
32  # Loop de envio de mensagens
33  while True:
34      mensagem = input(">")
35      if mensagem.lower() == "/quit":
36          socket_cliente.sendall(mensagem.encode())
37          print("Você saiu do chat.")
38          socket_cliente.close()
39          break
40      # Envia a mensagem codificada
41      socket_cliente.sendall(mensagem.encode())
```

**Servidor:** Cada cliente conectado recebe uma thread separada para o gerenciamento de mensagens. Uma função é executada em uma thread nova para cada cliente, onde ele pode enviar mensagens, que são tratadas pelo servidor conforme a necessidade (broadcast ou unicast). Permitindo a conexão simultânea de múltiplos usuários ao servidor e evita que um cliente bloqueie o acesso de outros.

```
# Função para recebimento de mensagens do cliente
def recebe_dados(sock_cliente, endereco):
    nome = sock_cliente.recv(50).decode() # Recebe o nome do cliente
    print(f"Conexão bem sucedida com {nome} via endereço: {endereco}")
    # Adiciona o cliente na lista com seu nome e socket
    lista_clientes.append((nome, sock_cliente))
    broadcast(f"{nome} entrou no chat!", sock_cliente)
    while True:
        try:
            mensagem = sock_cliente.recv(1024).decode()
            if mensagem.startswith("/"):
                # Comando Unicast: /nome para mensagem privada
                destinatario, msg = mensagem[1:].split(' ', 1)
                unicast(destinatario, msg, nome)
            elif mensagem == "/quit":
                remover(sock_cliente, nome)
                break
            else:
                broadcast(f"{nome} >> {mensagem}", sock_cliente)
        except:
            remover(sock_cliente, nome)
            break
```

## Aplicação para rodar na web com interface gráfica:

### Tecnologias Utilizadas:

- Backend: Flask, Flask-SocketIO e Python
- Frontend: HTML, CSS com TailwindCSS, JavaScript
- Comunicação em Tempo Real: SocketIO para mensagens em tempo real e atualização de lista de usuários conectados

### Descrição das Funcionalidades

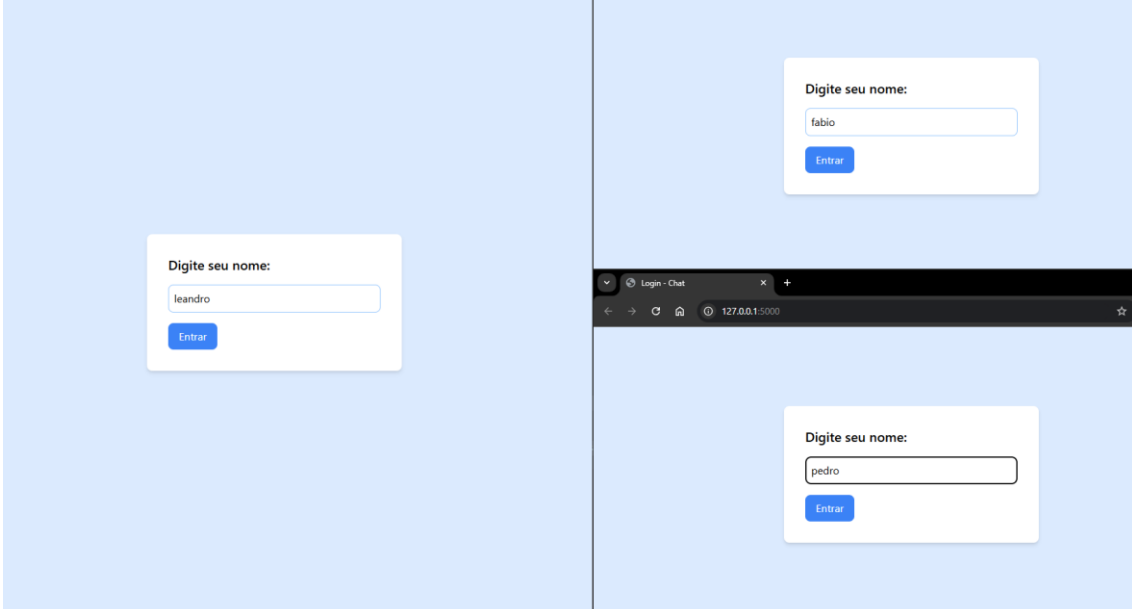
1. Página de Login (index.html):
  - Objetivo: Permitir ao usuário inserir seu nome para acessar o chat.
  - Comportamento:
    - Essa aba irá solicitar ao cliente o nome de exibição e enviar o nome codificado para o CHAT

Armazena o session ID e o nome do usuário para facilitar o gerenciamento e permitir mensagens privadas.

```
2. # Rota para ir para a página do chat
3. @app.route('/chat')
4. def chat():
5.     username = request.args.get('username')
6.     return render_template('chat.html', username=username)
```

- permite que o usuário insira seu nome. Ao clicar em "Entrar", a função enterChat() redireciona o usuário para /chat?username=nome, que é a página do chat

```
<script>
  function enterChat() {
    const username = document.getElementById('username').value;
    if (username) {
      window.location.href =
`/chat?username=${encodeURIComponent(username)}`;
    } else {
      alert("Por favor, digite seu nome.");
    }
  }
</script>
```



#### Página de Chat (chat.html):

- **Objetivo:** Interface de chat que permite mensagens públicas e privadas entre os usuários conectados.
- **Componentes:**

- Lista de Usuários: Exibe todos os usuários conectados em uma lista atualizada em tempo real, com opção de selecionar um destinatário para mensagens privadas.
- Área de Mensagens: Exibe as mensagens enviadas e recebidas, divididas em mensagens públicas e privadas.
- Campo de Entrada de Mensagem: Componente de input para o usuário digitar e enviar mensagens. As mensagens são enviadas para todos ou para um destinatário específico, dependendo da seleção.

### Função atualizar\_lista\_usuarios

Essa função emite a lista de todos os usuários conectados (usando socketio.emit) para que a interface do chat possa exibir uma lista atualizada.

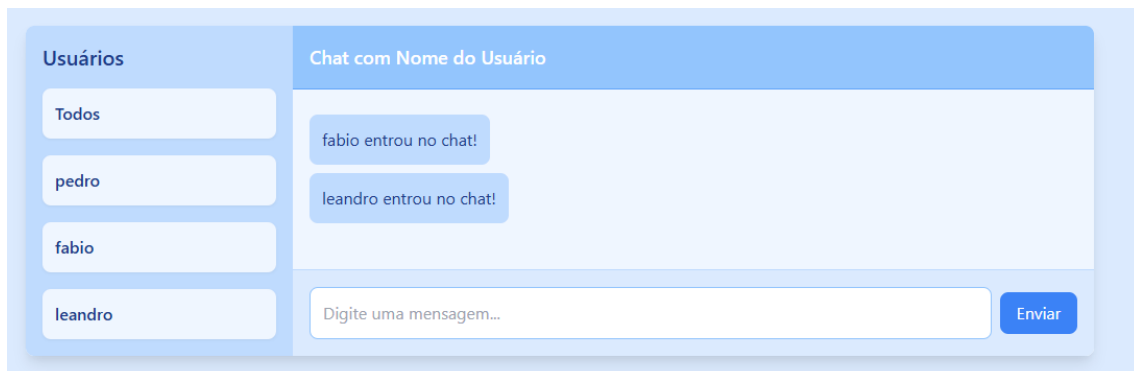
```
# Atualizar e emitir a lista de usuários para todos
def atualizar_lista_usuarios():
    user_list = list(clients.values())
    socketio.emit('update_user_list', user_list)
```

### Evento join

Esse evento é acionado quando um novo usuário entra no chat:

1. **Salva o usuário** no dicionário clients, mapeando request.sid (session ID) para o nome de usuário e atualiza a lista de usuarios para todos os clientes, chamando atualizar\_lista\_usuarios e notifica que tal usuário entrou no chat fazendo um BROADCAST

```
# Evento quando o usuário entra no chat
@socketio.on('join')
def handle_join(data):
    username = data['username']
    clients[request.sid] = username # Adiciona o usuário ao dicionário
    emit('message', {'msg': f"{username} entrou no chat!"}, broadcast=True, include_self=False)
    atualizar_lista_usuarios() # Envia a lista atualizada a todos
```



## Evento disconnect

Esse evento é chamado quando um usuário sai do chat:

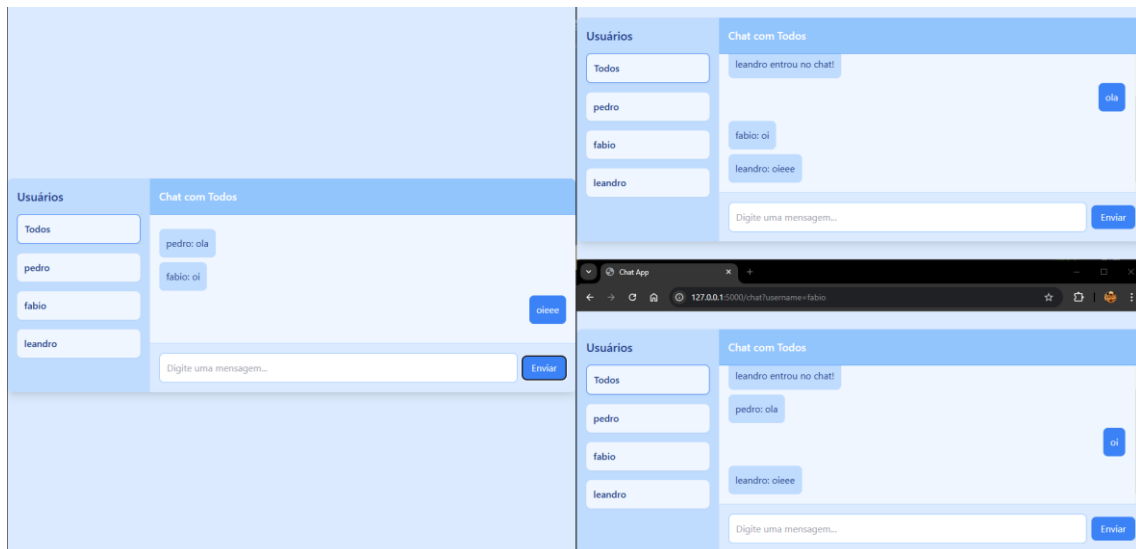
1. Remover o usuário do dicionário clientes e notifica a saída do usuário para todos e no final atualiza de usuários presentes no chat para todos.

```
# Evento quando o usuário sai do chat
@socketio.on('disconnect')
def handle_disconnect():
    username = clients.get(request.sid, "Unknown")
    if username:
        del clients[request.sid] # Remove o usuário do dicionário
        emit('message', {'msg': f"{username} saiu do chat!"}, broadcast=True, include_self=False)
        atualizar_lista_usuarios() # Envia a lista atualizada a todos
```

## Evento message

Este evento lida com mensagens públicas(usando BROADCAST). A mensagem é emitida para todos, menos para o remetente (se include\_self=False). Assim, todos veem a mensagem

```
# Evento de envio de mensagem pública
@socketio.on('message')
def handle_message(data):
    username = data['username']
    msg = data['msg'].strip() # Remove espaços em branco
    emit('message', {'msg': f"{username}: {msg}"}, broadcast=True, include_self=False)
```



## Evento private\_message

Este evento lida com mensagens privadas:

1. Busca o sid do destinatário no dicionário clients usando o nome e ao encontrar envia a mensagem privada para o destinatário encontrado usando room=recipient\_sid e no final valida se o destinatário foi encontrado ou não

```
2.
3. # Evento de envio de mensagem privada
4. @socketio.on('private_message')
```

```
5. def handle_private_message(data):
6.     sender = data['username']
7.     recipient_name = data['recipient']
8.     msg = data['msg']
9.     # Busca o `sid` do destinatário
10.    recipient_sid = None
11.    for sid, name in clients.items():
12.        if name == recipient_name:
13.            recipient_sid = sid
14.            break
15.    # Enviar mensagem privada para o destinatário
16.    if recipient_sid:
17.        # Envia a mensagem para o destinatário sem reenvio ao
        remetente
18.        emit('message', {'msg': f"(Privado de {sender}): {msg}"},
        room=recipient_sid)
19.        # Envia uma confirmação para o remetente de que a mensagem
        foi enviada
20.        emit('message', {'msg': f"(Privado para {recipient_name}):
        {msg}"}, room=request.sid, include_self=False)
21.    else:
22.        # Notifica o remetente que o destinatário não foi
        encontrado
23.        emit('message', {'msg': f"Usuário {recipient_name} não
        encontrado."}, room=request.sid)
```

