Projeto Final - Conectividade em Sistemas Ciberfísicos

A implementação do chat com sockets foi para estabelecer comunicação entre o servidor e os clientes, que seria (pessoa1, pessoa2, pessoa3) a mesma pessoa.
Ambos operam no TCP, garantindo uma conexão estável e detecção de desconexão nos clientes.

No **servidor**, a implementação do **socket**, com um exemplo do código:

```
self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.server.bind((host, port))
self.server.listen(5)
```

Está explicando a implementação de um socket TCP em socket.SOCK_STREAM com o endereço 127.0.0.1 para porta 200. No listen, está aguardando conexões de até 5 clientes simultâneos.

No caso do **broadcast**, deve ser usado para enviar mensagens a todos os clientes conectados, exceto o remetente. Igual ao exemplo à seguir:

```
def broadcast(self, message, sender=None):
    for client in self.clients:
        if client != sender:
            try:
                client.send(message.encode())
            except:
                self.remove_client(client)
```

Caso de falha no envio, o cliente é removido.

Para o envio de mensagem para todos os clientes conectados é utilizado self.clients e assim o servidor envia a mensagem desejada.

Também, se algum cliente estiver offline no servidor ou não puder receber mensagens, é removido, utilizando o remove_client.

Para enviar alguma mensagem privada para somente um cliente, deve-se usar o comando process_command e send_private.

No **cliente**, a implementação do socket com a conexão, usa o TCP que se conecta ao servidor, após a conexão, é enviado o nome do usuário. Um exemplo do código:

```
self.client = socket.socket(socket.AF_INET,  socket.SOCK_STREAM)
self.client.connect(('127.0.0.1',  200))
self.client.send(self.name_entry.get().encode())
```

Para o envio de mensagens, os clientes digitam e irá ser enviada ao servidor via self.client.send, como no exemplo abaixo:

```
def send_message(self):
    self.client.send(msg.encode())
```

E para o recebimento de mensagens, os clientes devem verificar regularmente se possui mensagens do servidor, como no exemplo abaixo:

```
def check_messages(self):
    message = self.client.recv(1024).decode()
```

No caso de **identificação** dos usuários pelo nome, que é enviado pelo cliente assim que se conecta ao servidor, é implementado o self.clients, associando o socket ao nome do usuário. Além disso, sempre que um usuário entra ou sai, o servidor notifica todos os clientes com a seguinte mensagem:

```
self.broadcast(f"{username} entrou no chat!")
```

Para a saída, o servidor remove o socket e informa os outros clientes:

```
self.broadcast(f"{username} saiu do chat!")
```

O **servidor** utiliza o método select.select para gerenciar múltiplas conexões ao mesmo tempo sem a necessidade de ter **múltiplas threads**. O select substitui o uso do threads. Como no exemplo abaixo:

```
readable, _, _ = select.select([self.server] + list(self.clients.keys()), [ ], [ ],  0.1)
```

Isso permite que o servidor tenha novas conexões e mensagens recebidas de maneira eficiente.

Também utilizamos o after e a biblioteca tkinter para executar tarefas tranquilas, que é semelhante ao uso de threads para algumas tarefas. Um exemplo:

```
def check_messages(self):
   try:
      self.client.settimeout(0.1)
      message = self.client.recv(1024).decode()

        except socket.timeout:
      pass
   except:

       self.root.after(100, self.check_messages)
```

O check_messages verifica frequentemente se há mensagens recebidas do servidor. Já o self.root.after permite que essa verificação aconteça sem bloquear a execução da interface gráfica, mantendo o programa funcionando.

Integrantes: Camilla Augusta Uber e Arthur Betoni

https://github.com/arthurbertoni/ciber-.git