



INSTITUTO FEDERAL DO ESPÍRITO SANTO
BACHAREL EM SISTEMAS DE INFORMAÇÃO

DAVID DE MOURA MARQUES

LOJA DE PRODUTOS ESPORTIVOS
Trabalho 2 - Protocolos com paralelismo

SERRA
2022

DAVID DE MOURA MARQUES

LOJA DE PRODUTOS ESPORTIVOS

Trabalho 2 - Protocolos com paralelismo

Trabalho apresentado no curso de
graduação do Instituto Federal do
Espírito Santo.

Orientadora: Dra. Cristina Klippel
Dominicini

SERRA
2022

Sumário

Objetivos	3
Descrição	3
Solução	3
Arquitetura	3
Servidor	5
Cliente	6
Formato de comunicação	7
Fluxo de comunicação	9
Multi-threading	10
Conclusão	10

Objetivos

Objetivo geral do trabalho é desenvolver uma aplicação cliente/servidor utilizando a biblioteca de programação socket com TCP na linguagem Python, versão 3.6, ou superior.

Os objetivos específicos do trabalho são:

- Familiarizar-se com a programação utilizando a API socket.
- Enviar e receber dados em uma aplicação que utiliza a arquitetura Cliente/Servidor.
- Entender o conceito de protocolos.
- Entender o funcionamento de aplicações concorrentes.

Descrição

Neste trabalho você será desafiado a melhorar a aplicação distribuída desenvolvida no Trabalho 1, que simula uma venda automatizada entre uma loja de produtos esportivos e seus consumidores. Obs.: Neste trabalho, não será necessário implementar o componente Fornecedor. Conforme as instruções apresentadas neste documento, agora será necessário suportar: uma especificação fixa para o formato das mensagens, autenticação de usuário, armazenamento de pedidos realizados e comunicação concorrente.

Solução

Afim de atender os requisitos solicitados, implementou-se uma aplicação python que faz uso de bibliotecas que dão suporte a API socket e aplicações concorrentes através de threads

A aplicação foi desenvolvida em inglês para manter uma universalização da mesma, entretanto os comentários foram escritos em português afim de facilitar o entendimento acadêmico.

Arquitetura

A aplicação foi projetada sob uma arquitetura onde cada entidade possui um módulo contendo seu modelo no formato de classe e seu repositório, responsável por manipular os arquivos csv das suas classes referentes. Ex. O módulo Product tem o model Product.py e seu repository, ProductRepository.py, como este, também temos os modulos Orders e Users.

Temos também o modulo common com o arquivo Utils.py que contém funções genéricas que podem ser usada por várias partes da aplicação.

Os dados da aplicação, armazenados em arquivos csv ficam localizados no diretório Data.

Temos também o Módulo Message que possui as classes que definem o protocolo e formado das mensagens que devem ser trocadas entre o servidor e clientes.

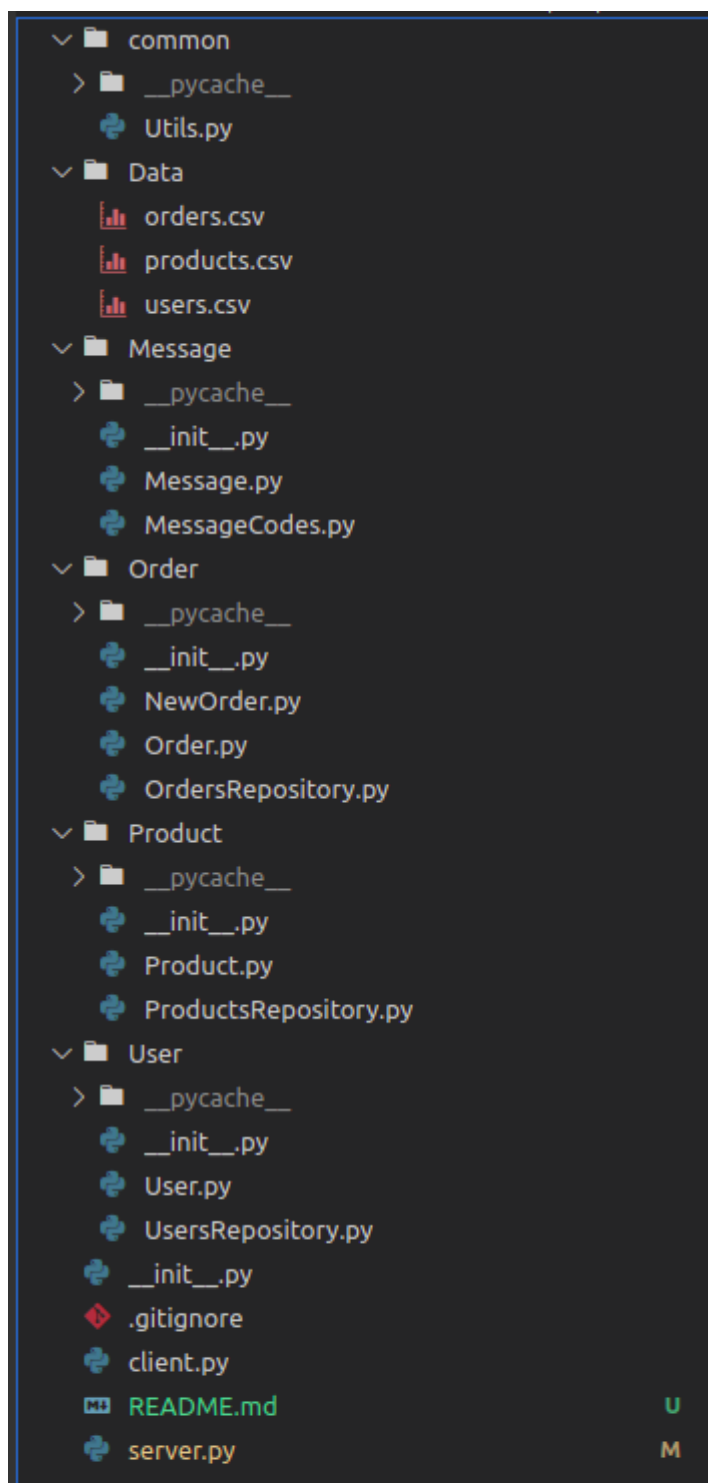


Figura 1. Arquitetura do sistema

Servidor

A loja de produtos esportivos está representada no arquivo `server.py` e tem a seguinte configuração:



```
1 def Main():
2     HOST = '127.0.0.1'
3     PORT = 3333
4
5     # carregando lista de usuários na memória
6     users = FindAllUsers()
7
8     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
9         s.bind((HOST, PORT))
10        print("socket binded to port", PORT)
11        s.listen()
12        print("socket is listening")
13
14    while True:
15        conn, addr = s.accept() # estabelece conexão com o cliente
16        print('Connected to: ', addr[0], ':', addr[1])
17        start_new_thread(threaded, (conn, users)) # cria uma nova thread e retorna seu id
18
```

Figura 2. Configuração do arquivo `Server.py`

O host a qual o servidor estará disponível é o 127.0.0.1, ou seja, localhost, através da loopback interface, a porta por onde escutará as requisições é a 3333, conforme especificado no documento de requisitos.

Os usuários da aplicação são carregados em memória através da função `FindAllUsers()` implementada no arquivo `UsersRepository.py`

Na linha 8 temos a seguinte instrução:

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
```

Neste momento estamos configurando nosso socket através da criação de uma instância que recebe dois parâmetros `socket.AF_INET`, que indica que será utilizado um IP v.4 e `socket.SOCK_STREAM` que informa que o protocolo utilizado será o TCP. Como estamos utilizando a instrução `with` não é necessário encerrar o socket explicitamente com `socket.close()` a aplicação se encarrega de realizar este

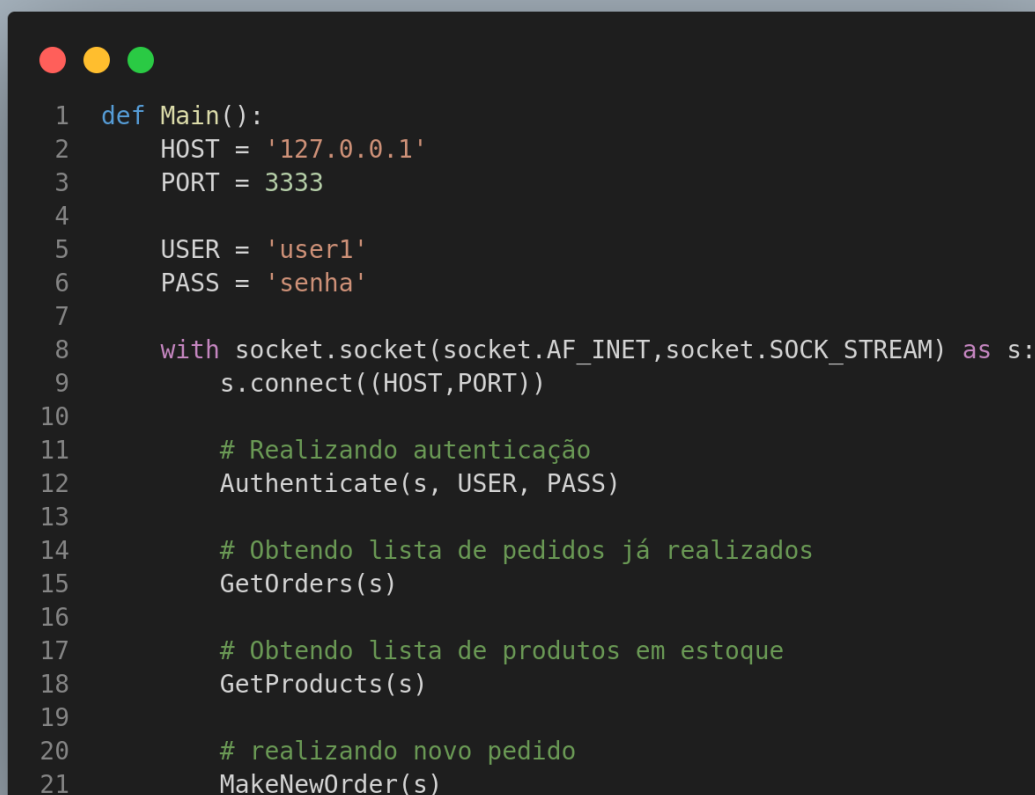
procedimento automaticamente. A partir deste ponto podemos utilizar a instância do socket através da variável `s`.

Na linha 11 ativamos o modo *listen* do socket e na linha 15 entramos em um loop aceitando novas conexões.

Ao receber uma nova conexão, uma thread é criada onde toda a requisição é tratada através da função `threaded()` que recebe como parametro a conexão e a lista de usuários.

Cliente

Os clientes são representados no arquivo `client.py` e tem a seguinte configuração:



```
1 def Main():
2     HOST = '127.0.0.1'
3     PORT = 3333
4
5     USER = 'user1'
6     PASS = 'senha'
7
8     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
9         s.connect((HOST, PORT))
10
11         # Realizando autenticação
12         Authenticate(s, USER, PASS)
13
14         # Obtendo lista de pedidos já realizados
15         GetOrders(s)
16
17         # Obtendo lista de produtos em estoque
18         GetProducts(s)
19
20         # realizando novo pedido
21         MakeNewOrder(s)
```

Figura 3. Configuração do arquivo `client.py`

Similar ao `server.py` eles também especificam um host e porta, porém neste caso estes são referentes ao servidor a qual desejam se conectar. Também possuem um usuário e senha para autenticação.

Esta aplicação se comunica com o servidor da seguinte forma:

1. Conecta-se ao servidor
2. Envia credenciais de acesso


3. Solicita lista de pedidos realizados anteriormente
4. Solicita lista de produtos disponíveis em estoque
5. Realiza novos pedidos
6. Encerra conexão

Sendo que o passo 1 e 2 são obrigatórios para qualquer aplicação e devem ser executados na mesma ordem apresentada acima.

As funções auxiliares apresentadas acima podem ser encontradas no mesmo arquivo `client.py`

Formato de comunicação

A Comunicação entre clientes e servidor é especificada através da classe `Message.py` que possui a seguinte estrutura:



```
1 class Message:
2     def __init__(self, messageCode, data, endOfData):
3         self.code = messageCode
4         self.data = data
5         self.endOfData = endOfData
```

Figura 4. Formato das mensagens

Ou seja, sempre deve ser recebido e enviado um objeto com esta estrutura que é composta pelos campos `code`, `data` e `endOfData`.

- **Code:** Indica o tipo de mensagem que está sendo enviada ou recebida, é especificado pelo enum `MessageTypes` e deve, obrigatoriamente ter um valor existente neste enum, que são
 - **AUTHENTICATION:** Mensagem de autenticação, deve ser enviado usuário e senha
 - **GET_PRODUCTS:** Solicitação da lista de produtos disponíveis em estoque
 - **PRODUCTS_LIST:** Indica que a mensagem contém uma lista de produtos
 - **GET_ORDERS:** Solicitação da lista de pedidos já realizados pelo usuário logado
 - **ORDERS_LIST:** Indica que a mensagem contém a lista de pedidos realizados pelo usuário logado
 - **NEW_ORDER_ITEM:** Solicitação de compra de um novo produto

- SUCCESS: Confirmação de sucesso
- ERROR: Indica ocorrência de erros durante o processamento

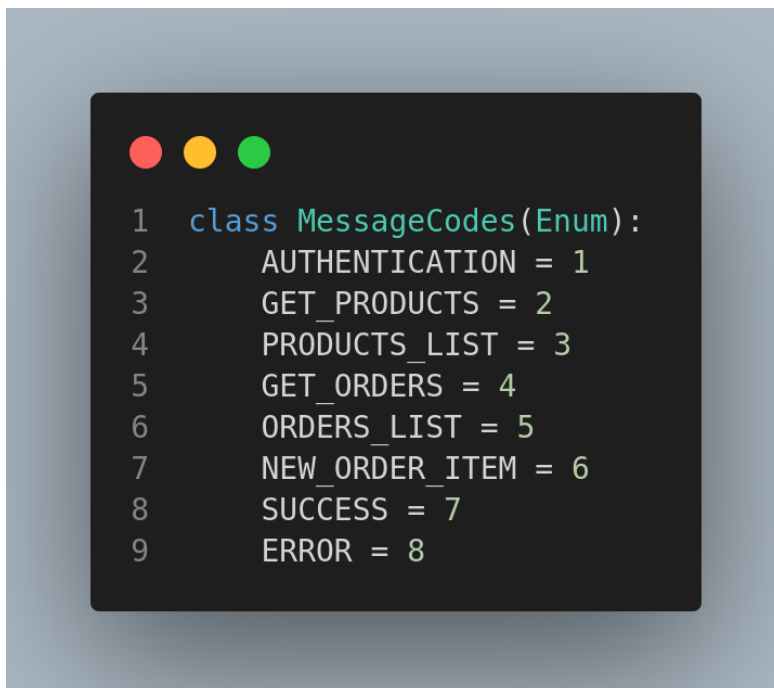


Figura 5. MessageCodes

- Data: Possui o corpo da mensagem, o dado que foi solicitado ou enviado pelo cliente ou servidor
- endOfData: Boleano que indica se a mensagem foi totalmente enviada ou se há novas mensagens a serem enviadas posteriormente, usada para realizar requisições de vários itens.

Utilizou-se a mesma classe na aplicação servidor e cliente, entretanto, em um caso onde cliente e servidor estão fisicamente distantes, será necessário duplicar esta interface de comunicação afim de mantermos um canal comum dos dois lados.

Fluxo de comunicação

O fluxo de comunicação é estabelecido na função `threaded()` e deve seguir o seguinte fluxo:

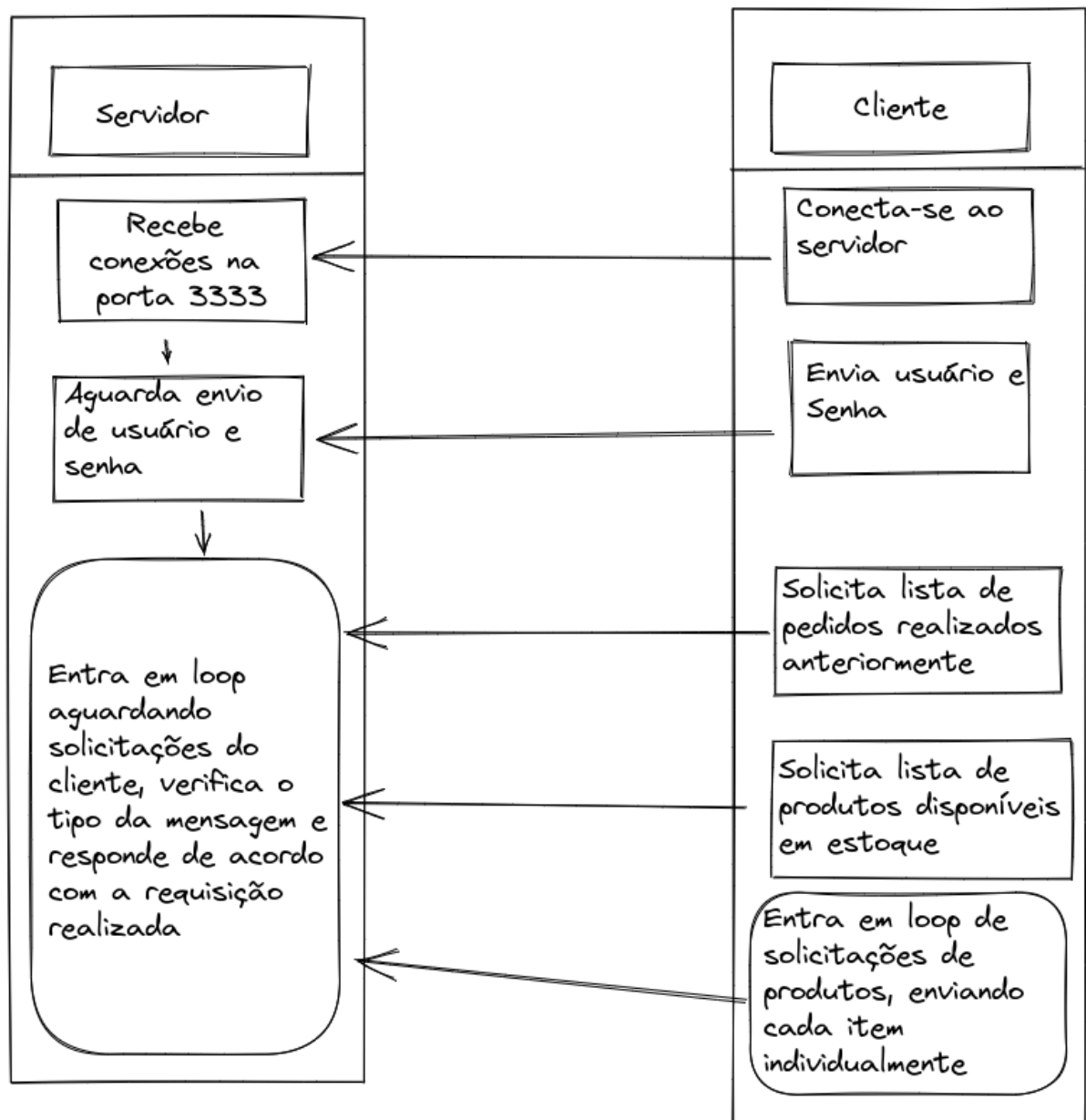


Figura 6. Fluxo de mensagens

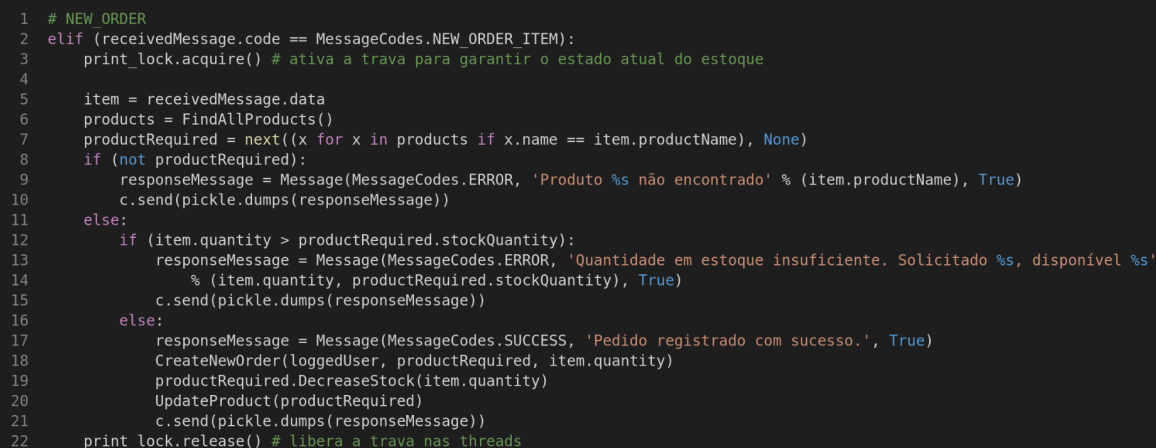
Como visto, após a autenticação, a ordem das solicitações pode variar, podendo, por exemplo, o cliente efetuar um pedido antes de solicitar a lista de pedidos realizados, ou mesmo, não realizar uma determinada solicitação.

O código do servidor responsável por gerenciar este fluxo pode ser encontrado no arquivo `server.py` na função `threaded()`.

Multi-threading

Como apresentado, a aplicação faz uso de threads para permitir múltiplas conexões simultâneas, isto gera problemas de concorrência aos arquivos compartilhados pois podemos ter mais de um cliente atualizando o mesmo arquivo ao mesmo tempo, para garantir a consistência na atualização dos dados e resolver este problema foi utilizado o recurso de trava das threads durante o processo de atualização dos arquivos compartilhados, entretanto, processos de leitura não fizeram uso desta trava afim de deixar o sistema mais fluido.

O código responsável pela escrita nos arquivos e que possui a trava das threads é apresentado a seguir e faz parte da função threaded



```

1  # NEW_ORDER
2  elif (receivedMessage.code == MessageCodes.NEW_ORDER_ITEM):
3      print_lock.acquire() # ativa a trava para garantir o estado atual do estoque
4
5      item = receivedMessage.data
6      products = FindAllProducts()
7      productRequired = next((x for x in products if x.name == item.productName), None)
8      if (not productRequired):
9          responseMessage = Message(MessageCodes.ERROR, 'Produto %s não encontrado' % (item.productName), True)
10         c.send(pickle.dumps(responseMessage))
11     else:
12         if (item.quantity > productRequired.stockQuantity):
13             responseMessage = Message(MessageCodes.ERROR, 'Quantidade em estoque insuficiente. Solicitado %s, disponível %s'
14                                     % (item.quantity, productRequired.stockQuantity), True)
15             c.send(pickle.dumps(responseMessage))
16         else:
17             responseMessage = Message(MessageCodes.SUCCESS, 'Pedido registrado com sucesso.', True)
18             CreateNewOrder(loggedUser, productRequired, item.quantity)
19             productRequired.DecreaseStock(item.quantity)
20             UpdateProduct(productRequired)
21             c.send(pickle.dumps(responseMessage))
22     print_lock.release() # libera a trava nas threads

```

Figura 7. Lock das threads para escrita em arquivos

Conclusão

Como resultado temos uma aplicação cliente x servidor baseada em sockets utilizando a tecnologia de threads que trouxe grandes aprendizados quanto aos protocolos de comunicação existentes e tecnologias disponíveis que possibilitam paralelismo. Os desafios ficaram por conta da linguagem, que não é do meu domínio e conceitos teóricos sobre as tecnologias utilizadas que requeriram grande tempo dedicado a estudos e entendimento de conceitos.

Por fim, apesar de todo o esforço, o sentimento de aprendizado é o que se mantém de forma que concluo que o trabalho acrescentou bastante nos meus conhecimentos como profissional.