Denilson H Silva

Cliente-Servidor UDP

Dourados, MS
Julho 2019

Sumário

1 Problema a ser tratado

Este documento tem como objetivo descrever a implementação de um cliente e um servidor que usam o protocolo UDP para sua comunicação. Além da implementação básica do que seria necessário para o funcionamento da comunicação, foram implementadas outras rotinas como o *checksum* e retransmissão de dados para garantir a troca atômica dos dados entre o cliente e o servidor. Também foi usando o conceito de *Threading* no servidor para que fosse possível a conexão de vários usuários ao mesmo tempo.

2 Decomposição da implementação

Aqui será discutido todos os pontos de implementação do código tanto do cliente como do servidor. O programa possui 4 arquivos: dados.json, comunicacao.py, servidor.py e cliente.py.

2.1 dados.json

Os dados trafegados entre o cliente e o servidor são do tipo JSON. O JSON permite que objetos sejam manipulados muito facilmente com ele. E em Python é fácil transformar um objeto em string e a string par objeto novamente, logo, é simples para ser transportado. Este arquivo já possui inicialmente dados salvos em JSON. Os dados dentro desse arquivo são uma lista com 9 registros, segue o exemplo de um registro:

```
{
    "tipoCombustivel": "0",
    "valor": 4000,
    "coordenadas": "(-22.217551;-54.765349)"
}
```

Foi usado 3 coordenadas distintas de 3 postos de gasolina na cidade de Dourados. Os postos são: Auto Posto Mitai (-22.217551;-54.765349), Auto Posto Ipiranga (-22.226945;-54.807717) e Auto Posto Tauros (-22.223665;-54.771388). Cada posto tem um tipo de combustível com o valor diferente um dos outros.

2.2 comunicacao.py

Esse arquivo é responsável por fazer a conexão e transmissão dos dados entre o cliente e o servidor. Esse arquivo é o *core* deste programa sendo o *end-point* entre o cliente e o servidor.

2.2.1 Novo pacote

Para criar um novo pacote chamar a função novoPacote():

```
def novoPacote(dados=None, SYN=False, ACK=False, FIN=False, ack=0, seq
    =0, proxSeq=0, checksum=None):
     return {
         "dados": dados,
         "SYN": SYN,
         "ACK": ACK,
         "FIN": FIN,
```

```
        "ack": ack,
        "seq": seq,
        "proxSeq": proxSeq,
        "checksum": checksum
    }
```

Essa função retorna um objeto já com valores padrões "setados", e para criar um pacote com alguma das chaves com um valor especifico para chamar a função passando este valor, como por exemplo:

```
novoPacote(ACK=True)
```

Cada pacote possui 8 campos e tanto o servidor quanto o cliente possuem a mesma estrutura de pacote. Aqui temos uma descrição para cada atributo do pacote:

- **dados:** Do lado do cliente é uma string contendo a ação a ser aplicada no servidor. Do lado do servidor, é a mensagem de resposta para o cliente informando se tudo ocorreu bem ou se ouve algum problema.
- **SYN:** Um *flag* de sincronização utilizada no momento do *Three-way Handshake*.
- **ACK:** Um *flag* que sinaliza que um ack está sendo enviado.
- **FIN:** Um *flag* que sinaliza que um dos lados está encerrando a conexão.
- **ack:** O valor do ack daquele pacote.
- **seq:** O número de sequência daquele pacote.
- **proxSeq:** O próximo numero de sequência esperado para aquele pacote.
- **checksum:** A soma dos bits daquele pacote.

2.2.2 Checksum

O *checksum* é calculado usando o algoritmo *Longitudinal Redundancy Check*. A função recebe o pacote e faz o calculo em cima do pacote sem o *checksum* existente nele. O retorno da função é a soma e um *boolean* que é gerado da comparação entre o calculo que acabou de ser feito com o valor que já estava no pacote:

```
return sum, sum == checksumDados
```

2.2.3 Receber Dados

A classe ReceberDados é responsável por ficar esperando por uma resposta e se a resposta não chegar em um dado tempo, ela fecha a conexão. Nela são implementadas três funções, tirando seu construtor: setInterval(), resetar() e escutar(). Na hora de criar uma instância é obrigatório que seja passado o socket que se quer escutar o recebimento de algo.

2.2.3.1 setInterval()

A cada 0.5 segundos esta função é chamada e ela pode ser chamada 50 vezes no máximo. Quem controla as chamadas é a função *Timer* da biblioteca *threading* do Python. Essa função ativa o *timer* sem bloquear o processo.

Se ela foi chamada por um cliente e o número de chamadas foi estourado, uma mensagem é mostrada para o cliente dizendo que o servidor não deu nenhuma resposta e o cliente é fechado. No caso do servidor, um pacote vazio é enviado dele para ele mesmo e como os dados do pacote não serão como os esperados por ele, a conexão com o cliente será encerrada.

2.2.3.2 resetar()

A tarefa principal desta função é encerrar o *timer* chamando sua função *cancel()*. Desta forma, quando existe ou não uma resposta, tanto do servidor quando do cliente, o *timer* é encerrado.

2.2.3.3 escutar()

Esta função é a única chamada pela instância da classe. Ela inicia chamando o setInterval() chama a função recvfrom() do socket para ficar ouvindo a resposta e quando a resposta chega, a função resetar() é chamada e os dados são retornados.

2.2.4 Conectar e aceitar

Funções responsáveis por fazer o *Three-way Handshake* entre o cliente e o servidor. A melhor maneira de explicar o funcionamento delas é mostrando no código.

2.2.4.1 conectar()

Função chamada pelo cliente para iniciar uma conexão:

```
1  def conectar(IP, PORT):
2      # Cria um novo pacote com a flag de sincronizacao ativada e se
       espera que o ack recebido seja 1
3      pacote = novoPacote(SYN=True, proxSeq=1)
4
5      sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
6      # Envia o pacote para o servidor
7      sock.sendto(json.dumps(pacote).encode(), (IP, PORT))
8
9      data, (ipServidor, portaServidor) = ReceberDados(sock).escutar()
10
11     resposta = json.loads(data.decode())
12
```

```
13      # Dos dados recebidos, se espera as flags ACK e SYN, e que o ack
    seja igual ao proxSeq do pacote enviado
14      if resposta["SYN"] and resposta["ACK"] and resposta["ack"] == pacote
    ["proxSeq"]:
15
16          # Caso o servidor retorne is dados esperados, e enviado um novo
    pacote com a flag ACK, o ack e o numero de sequencia
17          dadosDeConfirmacao = novoPacote(
18              ACK=True,
19              ack=resposta["seq"] + 1,
20              seq=1
21          )
22          sock.sendto(
23              json.dumps(dadosDeConfirmacao).encode(),
24              (ipServidor, portaServidor)
25          )
26          return sock, portaServidor
27      return False, resposta
```

2.2.4.2 aceitar()

Função chamada pelo servidor para aceitar uma conexão:

```
1  def aceitar(dadosCliente, numeroCliente, PORTA):
2      dados, ipCliente, portaCliente = dadosCliente
3      dados = json.loads(dados.decode())
4
5      if dados["SYN"]:
6          # Um novo socket sera criado para o novo cliente
7          # Cada cliente tera um thread vinculada a uma porta
8          porta = PORTA + numeroCliente
9          sockCliente = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
10         sockCliente.bind(("", porta))
11
12         # Um novo pacote eh criado para responder a solicitacao do
    cliente
13         pacote = novoPacote(
14             SYN=True, ACK=True,
15             proxSeq=1,
16             ack=dados["seq"] + 1
17         )
18         sockCliente.sendto(
19             json.dumps(pacote).encode(),
20             (ipCliente, portaCliente)
21         )
22
23         dados, (ipCliente, portaCliente) = ReceberDados(
24             sockCliente,
```

```
            SERVIDOR).escutar()
        resposta = json.loads(dados.decode())

        # Com a resposta do cliente, retornamos o novo socket que sera usado pela thread do cliente
        if resposta["ACK"] and resposta["ack"] == pacote["proxSeq"]:
            return sockCliente, (ipCliente, portaCliente)
    return False, (ipCliente, portaCliente)
```

2.2.5 Transmitir

Por fim, temos a função de transmissão e retransmissão que é usada pelo cliente para enviar seus dados. Ela começa fazendo o *checksum* do pacote, depois, envia o pacote para o servidor. Se a resposta do servidor for diferente da esperada, uma chamada recursiva é feita para esta mesma função e uma mensagem de quantas retransmissões já aconteceram é mostrada:

```
    if not pacoteRec['ACK'] or pacoteRec["ack"] != pacote["proxSeq"]:
        if contagem < N_MAX_RETRANSMISSOES:
            print("Retransmissao ({}/5)".format(contagem))
        return transmitir(
            sock,
            pacote,
            IP,
            PORTA,
            fazerTransmicao= True if contagem != N_MAX_RETRANSMISSOES else False,
            contagem = contagem + 1
        )
```

Se após 5 retransmissões nada acontecer as tentativas se encerram.

2.3 servidor.py

No começo do arquivo é analisado a quantidade de parâmetros passado para o servidor. No *main* do servidor, criamos um socket UDP e colocamos o socket para receber solicitação de conexão de algum cliente. Quando ele recebe algo, uma *thread* é criada para aquele cliente. A seguir será feita a descrição para o resto das funções, que valem apena falar sobre, neste arquivo.

2.3.1 comandoInvalido()

Uma função simples que checa se o comando mandado pelo cliente atende a todos os requisitos.

2.3.2 gravarDados()

Função responsável em pegar os dados enviados do cliente, transforma-los em um objeto e gravar no arquivo dados.json.

2.3.3 haversineFormula()

Uma função que recebe 4 parâmetros: Latitude e longitude que o cliente está buscando e a latitude e longitude de um posto salvo na base de dados. Cada passo do código está comentada para o melhor entendimento de como ela funciona:

```
def haversineFormula(latI, lonI, lat, lon):
    """
        Essa formula ira calcular a distancia entre dois pontos na terra
        (com lon e lat especificados em graus decimais). Os calculos sao
    baseados na terra circular, desconsiderando os efeitos elipsionais
    """

    # convertendo os graus decimais para radianos
    latI, lonI, lat, lon = map(radians, map(float, [latI, lonI, lat, lon
    ]))

    # aplicando a formula
    dlon = lon - lonI
    dlat = lat - latI

    # Quadrado da metade do comprimento do acorde entre os pontos
    a = sin(dlat / 2) ** 2 + cos(latI) * cos(lat) * sin(dlon / 2) ** 2

    # Distancia angular em radianos
    c = 2 * asin(min(1, sqrt(a)))

    R = 6371  # Raio da terra em quilometros

    return R * c
```

2.3.4 buscarResultado()

Retorna o posto com o menor valor de combustível dento do raio passado. Começa filtrando da base de dados os dados com o tipo de combustível requisitado, depois, filtra dentre esses dados todos os pontos que estão dentro do raio e, por fim, retorna dentre os últimos dados filtrados o que possuí o menor preço.

2.3.5 buscarDados()

A função começa buscando todos os dados da base e, com o comando enviado pelo cliente, é buscado um posto. Se algum é encontrado, uma mensagem formata é retorna para o cliente. Um exemplo de mensagem é:

```
Tipo combustivel: gasolina
Preco: R$ 4.001
coordenadas: (-13.23422; -22. 23423)
```

2.3.6 executar()

Está função é muito intuitiva. Nela, baseada no tipo de ação que o cliente deseja aplicar, será chamada a função responsável por gravar dados ou a função responsável por buscar os dados.

2.3.7 log()

Uma função simples que coloca no terminal tudo que está acontecendo no servidor. Existem 4 tipos de logs que são mostrado: Para quando um novo cliente é conectado, para quando o pacote é recebido, para quando existe um erro ao tentar se conectar com o cliente e para quando um cliente é desconectado.

2.3.8 processarConexao()

Esta é a principal função no servidor. Ela começa aceitando a conexão do cliente e entra em um *looping* para que um cliente possa se comunicar com o servidor por quanto tempo desejar. Dentro do *looping* é feito checagens de verificações dos pacotes que estão chegando do cliente. Explicando as partes mais importantes temos:

Verificação para saber se o número de sequência que acabou de chegar já foi enviado antes:

```
foiSeqClienteRecebido =  pacoteCliente["seq"] in seqRecebidos
```

Caso o número de sequência não tenha sido registrado até aquele dados momento, é executado todo o processo de execução do comando do cliente:

```
if not foiSeqClienteRecebido:
    comando = pacoteCliente["dados"]

    # Bloqueando thread para acessar a zona critica do servidor
    lock.acquire()
    resultado = executar(comando)
    lock.release()

    seqRecebidos.append(pacoteCliente["seq"])
```

Logo após a condição anterior ter sido executada, fazemos o cálculo do *checksum*, criamos uma novo pacote de resposta e o enviamos para o cliente:

```
1 sum, sumIguais = checksum(pacoteCliente)
2 pacote = novoPacote(
3     dados=resultado,
4     ack=pacoteCliente["seq"] + 1,
5     ACK=sumIguais,
6     checksum=sum
7 )
8 sock.sendto(json.dumps(pacote).encode(), (ipCliente, portaCliente))
```

Na linha 5 do código acima a flag ACK está recebendo o resultado da verificação entre o *checksum* do pacote recebido com o que acabou de ser calculado. Se eles não forem iguais o cliente saberá, desta forma, que ele tem que retransmitir os dados.

Se o servidor recebe um pacote do cliente com a flag FIN ativa, ele fecha a conexão com aquele cliente.

2.4 cliente.py

O cliente começa checando o número de parâmetros que foi passado. No *main*, um socket é criado e a conexão com o servidor é feita.

```
1 sock, resp = conectar(UDP_IP, UDP_PORT)
2 if sock:
3     porta = resp
4     print("Conexao estabilizada\n")
5     enviarDados(sock, UDP_IP, porta)
6 
7     # Finalizando a conexao
8     pac = novoPacote(FIN=True)
9     sock.sendto(json.dumps(pac).encode(), (UDP_IP, porta))
```

A variável resp, na linha 1 do código acima, possui o número da porta retornada pelo servidor. É por está porta que o cliente deverá conversar com o servidor, deste ponto em diante. O cliente possui mais duas funções para auxiliar seu funcionamento.

2.4.1 tratarEntrada()

Essa função pega o comando que o cliente está prestes a enviar e verifica todos os parâmetros batem com o esperado. Caso estejam, um objeto é preparado para ser enviado ao servidor.

2.4.2 enviarDados()

Função responsável por receber o comando do cliente e envia-los ao servidor. Primeiro montamos um pacote e o transmitimos para o servidor:

```
pacote = novoPacote(dados=dados, seq=seq, proxSeq = seq + 1)
resp = transmitir(sock, pacote, IP, PORTA)
```

Depois, se tudo ocorreu como o esperado e ouve uma resposta do servidor, mostramos os dados recebidos do servidor para o cliente:

```
respDados = resp["dados"]
del resp["dados"]
print("\nPacote recebido: {}".format(json.dumps(resp, indent=2)))
if dados["acao"].lower() == 'p':
    print("\nDados do posto encontrado")
    print("------------------------------")
print(respDados)
print("\n")

auxPacote = pacote
```

3 Entradas e como rodar

Este programa foi feito em Python, que por padrão é uma linguagem interpretada. O requisito mínimo para rodar, tanto o cliente quanto o servidor, é ter o Python na versão 3 ou superior. Para levantar o servidor basta, no terminal, executar o comando:

```
>> python3 servidor.py <PORTA>
```

Onde <PORTA> é o número da porta em que se deseja que o servidor seja levantado. Para levantar o cliente basta, no terminal, executar o comando:

```
>> python3 cliente.py <IP> <PORTA>
```

Onde <IP> é o endereço IP do servidor e <PORTA> é a porta em que o servidor foi aberto.

Vale ressaltar que é possível compilar o Python, com a intensão de proteger o código, usando alguma API como o Cython. Essa medida não será tomada aqui afim de evitar complicações futuras.

3.1 Comandos do cliente

Existem dos tipos de comandos que podem ser dados pelo cliente: Um de busca e outro de inserção. Cada um dos comandos tem 5 parâmetros. Na hora de ser inserido, devesse colocar um parâmetro seguido de um espaço e depois outro parâmetro. Segue um exemplo:

```
Entrada: D 1 4000 -22.2313 -23.13123
```

3.1.1 Tipos de dados

Cada tipo de comando deverá ser inserido na ordem que será descrito a seguir.

Comando de inserção:

- **Ação:** Para especificar que irá acontecer a inserção de um posto o primeiro parâmetro deve ser a letra **D**. Tanto faz se é maiúscula ou minuscula.
- **Tipo de combustível:** Um inteiro. Pode assumir 3 valores diferentes: 0 para diesel, 1 para álcool e 2 para gasolina.
- **Preço:** Um inteiro. O preço será salvo como múltiplo de 1000, então, se você deseja que o preço seja R$ 4, 99, por exemplo, o inteiro que devera ser passado é 4990.
- **Latitude:** Um número real. A latitude do posto.
- **Longitude:** Um número real. A longitude do posto.

Comando de busca:

- **Ação:** Para especificar que irá acontecer a busca de um posto o primeiro parâmetro deve ser a letra **P**. Tanto faz se é maiúscula ou minuscula.
- **Tipo de combustível:** Um inteiro. Pode assumir 3 valores diferentes: 0 para diesel, 1 para álcool e 2 para gasolina.
- **Raio:** Um inteiro. Indica qual deve ser o raio mínimo, em quilômetros, que um posto deve estar para ser retornado.
- **Latitude:** Um número real. A latitude do centro de busca.
- **Longitude:** Um número real. A longitude do centro de busca.

Apenas para ser o mais claro possível, se eu desejo por exemplo, encontra um posto com a gasolina mais barata dentro de um raio de 5 KM usando o centro de busca -22.4323;-23.3123, basta colocar na entrada:

```
1 Entrada: P 2 5 -22.4323 -23.3123
```

4 Testes

Aqui será mostrado a saída de vários testes feitos, com *print screens*.

4.0.1 Erguendo o servidor na porta 5000:

```
→   toSend git:(master) ✗  python3 servidor.py 5000
Servidor aberto na porta: 5000
```

4.0.2 Erguendo o cliente no endereço e porta do servidor:

```
→   toSend git:(master) ✗  python3 cliente.py 10.0.0.129 5000
Conexão estabilizada

Entrada:
```

Figura 1 – Informações no cliente

```
→   toSend git:(master) ✗  python3 servidor.py 5000
Servidor aberto na porta: 5000
Cliente conectado: 10.0.0.129:35724 - THREAD: 140525357856512
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figura 2 – Informações no servidor

4.0.3 Inserindo dados:

Figura 3 – Entrando com os dados e obtendo resposta do servidor

Figura 4 – Pacote do cliente quando chega no servidor

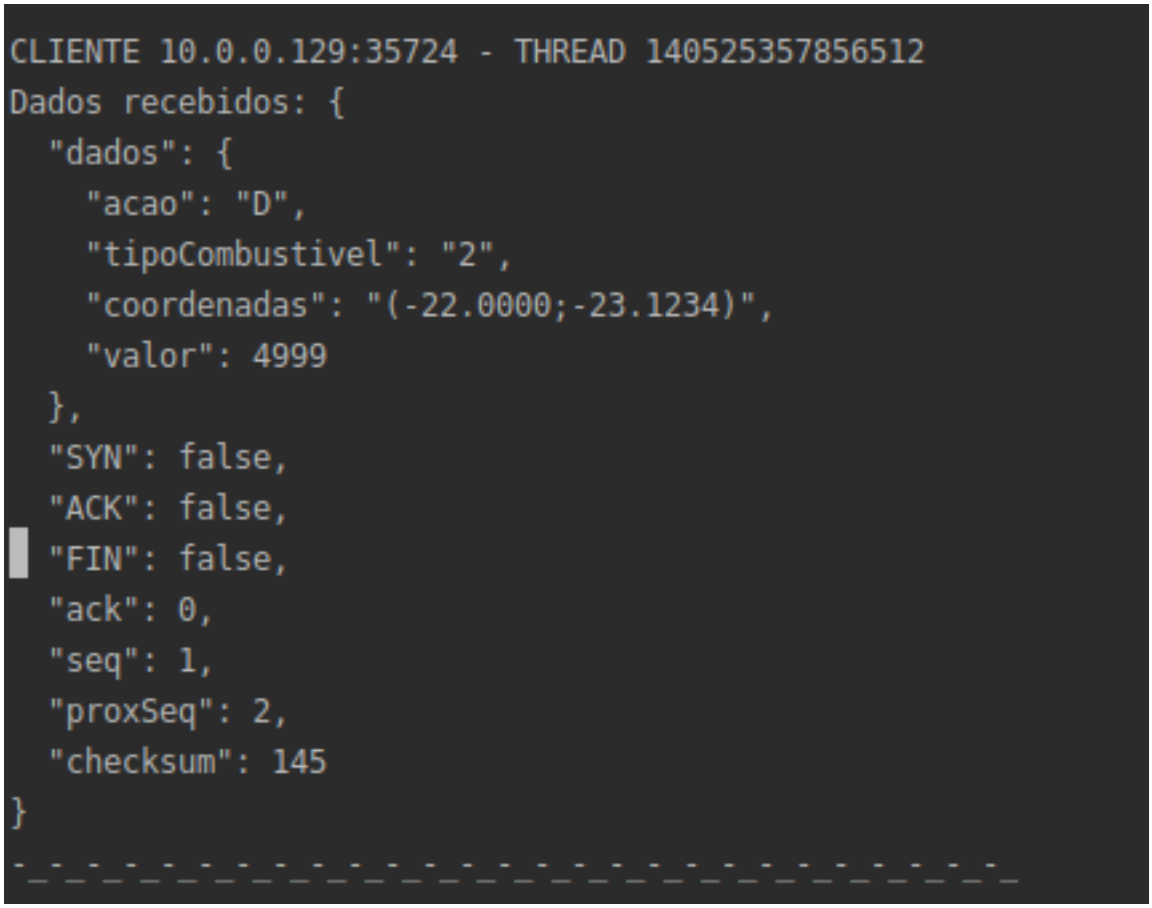
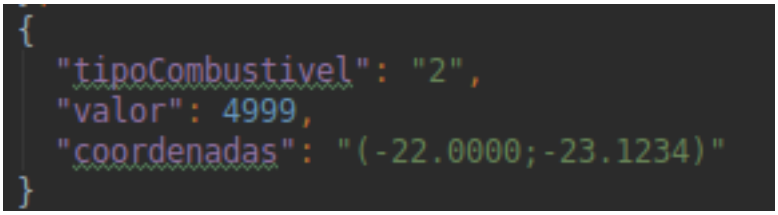


Figura 5 – Dado inserido na base



4.0.4 Buscando dados:

Figura 6 – Entrando com os dados de busca e obtendo resposta do servidor

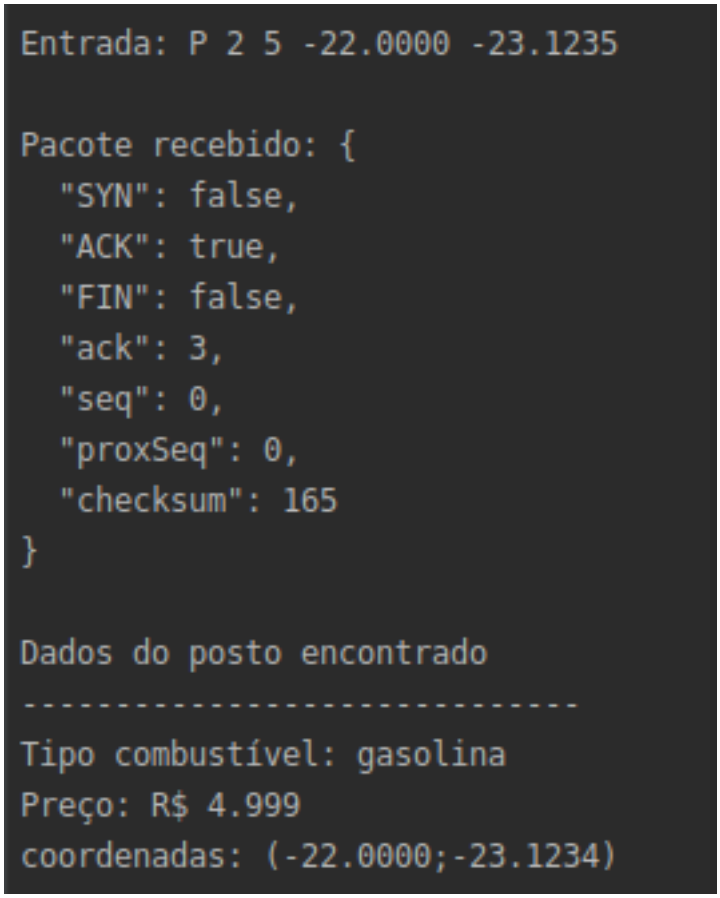
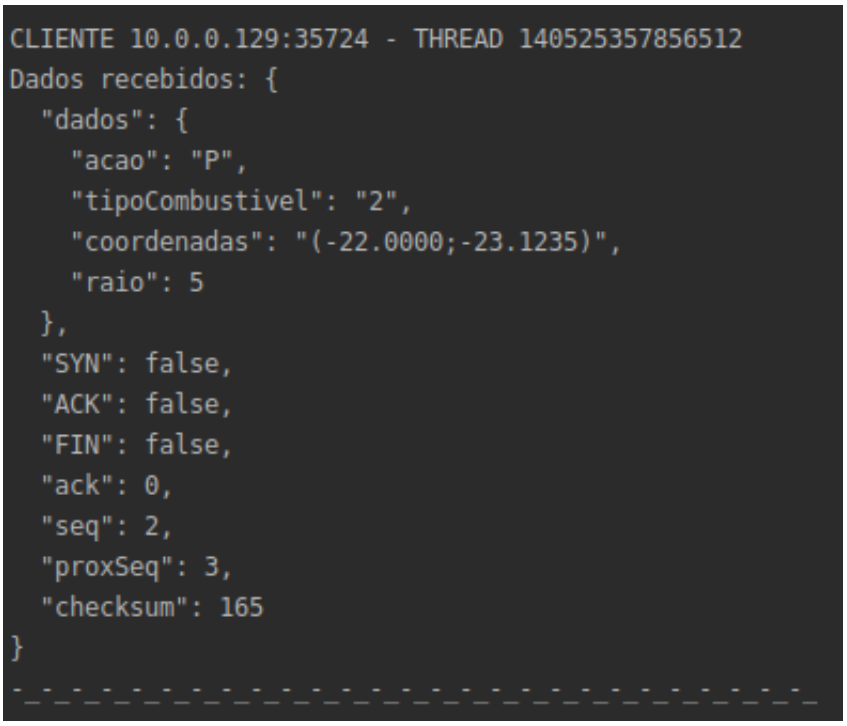


Figura 7 – Pacote do cliente quando chega no servidor



4.0.5 Tentando conexão com servidor:

```
Tentando conexão com o servidor (1/5)
Tentando conexão com o servidor (2/5)
Tentando conexão com o servidor (3/5)
Tentando conexão com o servidor (4/5)
Tentando conexão com o servidor (5/5)

Nenhuma resposta do servidor, tenha certeza que o mesmo está operando. Este cliente será encerrado
```

4.0.6 Retransmissão:

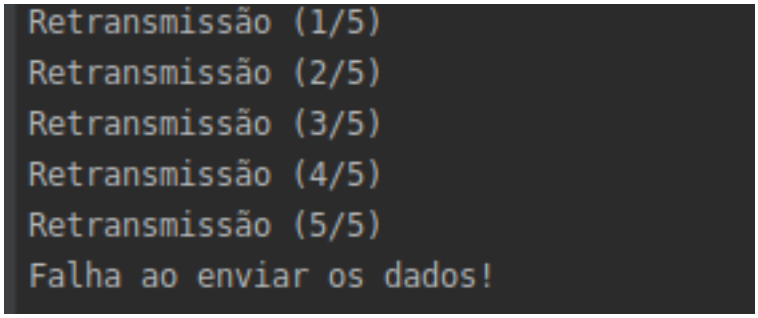


Figura 8 – logs de retransmissão no cliente

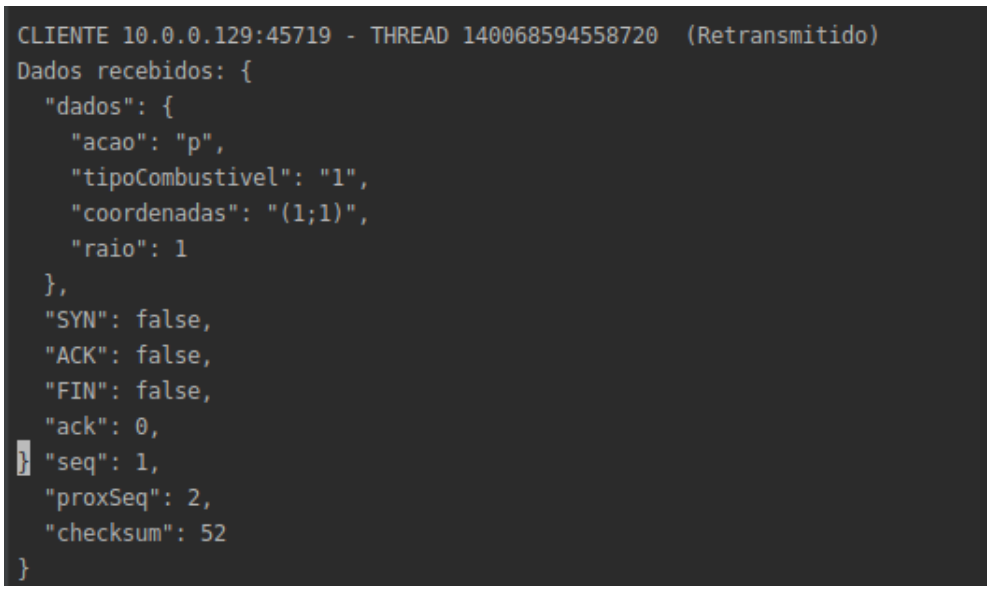
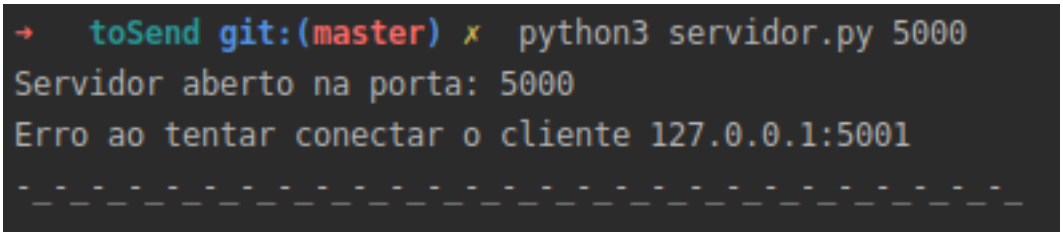


Figura 9 – Um pacote retransmitido pelo cliente no servidor

4.0.7 Cliente não termina o Handshake:



5 Conclusão

O que se pode tirar desse trabalho é que o objetivo foi concluído e as especificações foram aplicadas. A ideia principal era simular o básico do protocolo TCP e isso foi feito aplicando algoritmos de retransmissão e *checksum*. Muito ainda pode ser melhorado, obviamente, mas até aqui tudo foi uma grande fonte de aprendizagem e no é o que importa.

Referências

[geeks 2017]GEEKS, G. for. Socket programming in python. 2017. Disponível em: <https://www.geeksforgeeks.org/socket-programming-python/>.

[geeks 2017]GEEKS, G. for. Socket programming with multi-threading in python. 2017. Disponível em: <https://www.geeksforgeeks.org/socket-programming-multi-threading-python/>.

[Python 2019]PYTHON, D. socket — low-level networking interface. 2019. Disponível em: <https://docs.python.org/3/library/socket.htmlsocket.socket.listen>.

[Python 2019]PYTHON, D. threading — thread-based parallelism. 2019. Disponível em: <https://docs.python.org/3/library/threading.html>.

[Scripts 2019]SCRIPTS, M. T. Calculate distance, bearing and more between latitude/longitude points. 2019. Disponível em: <https://www.movable-type.co.uk/scripts/latlong.html>.