TP1 - Redes de Computadores- Turma TE - Lucas Prado Milhorato e Vitor Marques de Souza

A dupla optou por refazer o código herdado do TP0 para comunicação socket TCP/IP na linguagem python, onde encontramos o primeiro desafio que foi encontrar as libs e funções necessárias para adaptar a comunicação cliente servidor.

Ao refazer todo o código, percebemos que o python por ser uma linguagem de alto nível faz com que a mesma sintaxe fosse implementada de maneira muito mais legível e compacta além de fornecer várias funções que facilitam o trabalho de leitura, envio e recebimento de pacotes, por exemplo a função pack e unpack.

Focando agora nas particularidades do TP1, o primeiro passo dado foi interpretar como seria realizada a montagem dos quadros e a aplicação das regras de envio ACK e END. Para a montagem do quadro definimos uma classe que receberia as seguintes propriedades, demonstradas na figura 1 e definimos também uma segunda classe que define o input colocado pelo usuário:

```
class Frame:
    def __init__(self, sync1=0, sync2=0, length=0, checksum=0, id_=0, flags=0, dados=0):
        self.sync1 = sync1
        self.sync2 = sync2
        self.length = length
        self.checksum = checksum
        self.id_ = id_
        self.flags = flags
        self.dados = dados
class Configs:
    def __init__(self, id_flag=0, port_=0, input_=0, output_=0, ip_=0):
        self.id_flag = id_flag
        self.port_ = port_
        self.input_ = input_
        self.output_ = output_
        self.port_ = port_
        self.ip_ = ip_
```

Figura 1: Classes definidas

Para recebermos um input de um arquivo dividimos todos os caracteres do texto em uma lista de 7 em 7, visto que assim teremos um tamanho dinâmico para leitura de dados. Caso a mensagem não tenha 7 caracteres adicionamos espaços ao fim.

```
def send_msg_mult_pckg(s, config):
    totalsent = 0
    id_ = 0
    # input_ = open(f'{config.input_}.txt')
    chunk_size = int(7)
    msg = []
    with open(f'{config.input_}.txt') as fh:
        while (contents := fh.read(chunk_size)):
            count = 0
            if len(contents) < 7:
                while (7 - len(contents)):
                    contents = contents[:len(contents)] + ' '
                    print(contents, len(contents), 7 - len(contents))
                    count += 1
            msg.append(contents)
```

Figura 2: Enviando a mensagem de 7 em 7

Após receber o input do usuário, codificamos o dado recebido em bytes para string, setamos a flag para 0 e calculamos o checksum com as seguintes funções:

```python
def checksum(data):
    s = 0
    for i in range(0, len(data), 4):
        s += int(data[i:i+4], 16)
        if len(hex(s)) > 6:
            s = (int(hex(s)[2], 16) + int(hex(s)[3:], 16))
    return s ^ 0xFFFF

def fill_checksum(frame):
    cod = f'!IIHIBB{frame.length}s'
    # Empacota
    print('cheksum antes: ',frame.checksum)
    sdata = struct.pack(cod, frame.sync1, frame.sync2, frame.length, frame.checksum, frame.id_, frame.flags, frame.dados)
    print('sdata antes: ',sdata)
    chksum = checksum(binascii.b2a_hex(sdata).decode())
    print('checksum depois: ',chksum)
    sdata = struct.pack(cod, frame.sync1, frame.sync2, frame.length, chksum, frame.id_, frame.flags, frame.dados)
    print('sdata depois:', sdata)
    return sdata
```

Figura 3: Definição do checksum

As maiores dificuldades encontradas ao calcular o checksum foram relacionadas a estruturação de seu cálculo visto que envolve a soma de 4 bits juntamente do carrier digit.
De acordo com a figura acima é possível também perceber que definimos a estrutura e tipo dos dados do quadro do Frame f na variável cod, o que foi de certa maneira um desafio, visto que as codificações e funções requerem um certo tipo de dados para serem realizadas e muitas vezes durante o desenvolvimento não entendíamos se era um string, byte ou inteiro retornado e como realizaríamos a conversão em python.
Após empacotar o quadro o convertemos para a base 16 e enviamos para o outro lado da conexão.
Com a função unpack nativa do python temos acesso aos dados do quadro que haviam sido enviados de maneira compactada para sua interpretação e modificação. Segundo as instruções do TP ao receber o quadro devemos conferir o checksum e o id recebidos, além da correta estrutura dos dados que devem estar em formato DCC NET.
Para conferir o id checamos no envio do ACK, aproveitamos o bloco try/except que capta erros de recebimento e levantamos uma exceção caso o id enviado pelo ACK seja diferente do enviado ao servidor. Para conferir o checksum recalculamos o mesmo com os dados recebidos e comparamos com o que foi enviado.

```python
def check_chksum(frame):
    aux = frame.checksum
    frame.checksum = 0
    aux2 = fill_checksum(frame).checksum
    if aux == aux2:
        return True
    else:
        return False
```

Figura 4: Função que checa o checksum

```
try:
    rdata = s.recv(1024)
    rdata = base64.b16decode(rdata)
    unmtpacket = frame_unmount(rdata)

    if not check_chksum(unmtpacket):
        totalsent = totalsent - 1
        raise TypeError('id dif')

    if unmtpacket.id_ != id_:
        totalsent = totalsent - 1
        raise TypeError('id dif')
except:
    totalsent = totalsent - 1
    continue
```

Figura 5: Bloco Try e Except do recv

Caso o checksum e id esperados estejam corretos enviamos um quadro de confirmação ACK que consiste da mesma estrutura do quadro anterior porém com a flag inversa à recebida e sem nenhum dado. Além disso devemos enviar um novo quadro com a estrutura END ao fim da conexão, definimos suas chamadas da seguinte forma, zeramos os dados no seu recebimento assim enviamos o ACK e caso a flag da struct Config definida na figura 1 seja end, enviamos o END.

```
def frame_mount(config, id_=0):
    data_frame = config.input_
    cod = f'!IIHBBB7s'

    if config.id_flag == 'end':
        f = Frame(0xdcc023c2, 0xdcc023c2, 0, id_, 0, 0x40, str.encode('1234567'))
        ENDdata = struct.pack(cod, f.sync1, f.sync2, f.length, f.checksum, f.id_, f.flags, f.dados)
        # print(ENDdata)

        return ENDdata
    if data_frame == 0:
        f = Frame(0xdcc023c2, 0xdcc023c2, 0, id_, 0, 0x80, str.encode('1234567'))
        ACKdata = struct.pack(cod, f.sync1, f.sync2, f.length, f.checksum, f.id_, f.flags, f.dados)
        # print(ACKdata)

        return ACKdata
```

Figura 6: Condições de envio ACK e END