



Trabalho prático 1

Redes de computadores

Aluno: Eduardo Ávila Vilar
Matrícula: 2017001796

17/10/2022

Universidade Federal de Minas Gerais

— · —

Instituto de Ciências Exatas
Ciência da Computação

Relatório do trabalho prático 1

Primeiro Relatório da disciplina de Redes de Computadores do Departament de Ciência apresentado no curso de Engenharia de Sistemas da Universidade Federal de Minas Gerais.

Aluno: Eduardo Ávila Vilar

Professor: Dorgival Olavo Guedes Neto

Outubro
2022

Sumário

1	Introdução	1
2	Implementação	2
2.1	Funções de byte stuffing	2
2.2	Funções de checksum	3
2.3	Função para enviar ACK	5
2.4	Função para extrair o payload de um quadro	6
2.5	Função send	6
2.6	Função recv	7
3	Teste	9
3.1	Byte stuffing	9
3.2	Checksum	10
3.3	ACK	11
4	Conclusão	15

1 Introdução

O trabalho prático 1 da disciplina de Redes de Computadores consiste em implementar, em Python, a comunicação entre dois soquetes usando o Protocolo Ponto-a-Ponto (PPP), usando um *checksum* de 16 bits e realizando *byte stuffing* para escapar caracteres especiais dentro da mensagem. Este *byte stuffing* deve ser feito com o byte 0x7D, além de modificar o byte adiante, mudando os bytes 0x7D e 0x5D para 0x7E e 0x5E, respectivamente. É necessário também, que cada pacote seja reconhecido como recebido pelo receptor, ou seja, este tem que enviar de volta ao transmissor uma mensagem de reconhecimento (ACK) e no caso dela não chegar, o pacote em questão deve ser enviado novamente após 5 segundos. O próximo pacote só deve ser enviado depois do reconhecimento do envio do último pacote.

O pacote tem que ter 6 ou 7 campos, a depender se é um pacote do tipo ACK ou não. No caso de pacotes de mensagens são necessários: um campo de 1 byte de *flag* (0x7E) no início, para indicar que um novo quadro está começando; em seguida, um campo de 1 B de endereço que será sempre igual a 0xFF, pois entre duas máquinas não há necessidade de identificá-las; depois, um campo de 1 B de controle para diferenciar quadros do tipo ACK (0x03) de mensagens normais (0x07); após, um campo de 2 bytes de protocolo usado para enumerar os quadros, para garantir que as mensagens serão enviadas em ordem e confirmadas em ordem; um campo de até 1500 B de *payload* onde a mensagem real será inserida; um campo de 2 B de *checksum*, para identificar mensagens com erros; e finalmente, outro campo de 1 B de *flag*. O campo de *payload* é inexistente na mensagem de ACK, esta tendo, portanto, apenas 6 campos. Uma forma mais clara de entender o quadro é consultando a figura 1.



Figura 1: modelo de quadro.

Para isso, foram entregues 4 arquivos: *envarq.py*, onde o soquete do transmissor é montado; *recarq.py*, onde o soquete do receptor é montado; *pppsrt.py*, o único arquivo que deve ser modificado e onde as funções de enviar e receber do soquete serão implementadas para cumprir os requisitos pedidos na tarefa; e o arquivo *dcc023_tp1.py*, encarregado de encapsular a interface dos soquetes. Assim, uma esquema de camadas surge, pois *envarq.py* chama o *pppsrt.py* para inserir todos os campos descritos acima, este, por sua vez, invoca o *dcc023_tp1.py* para enviar os arquivos. Do outro lado, este mesmo programa recebe os arquivos, passa para o *pppsrt.py* do receptor para tirar o *byte stuffing* e checar se o *checksum* é o esperado. Se tudo estiver correto, ele envia um ACK de volta e passa a mensagem final para que *recarq.py* possa colocar em seu disco.

2 Implementação

2.1 Funções de byte stuffing

No início da classe PPPSRT, a única classe do arquivo *pppsrt.py*, todas as constantes citadas no capítulo anterior foram iniciadas como atributos da classe. Além disso, um dicionário foi criado para guardar as mensagens recebidas na função `recv` e uma variável inteira chamada `max_tries`, que indica quantas vezes um quadro deve tentar ser enviado antes de parar toda a operação.

```
26 from asyncio.format_helpers import extract_stack
27 from asyncore import read
28 from secrets import token_bytes
29 from typing import final
30 import dcc023_tpl
31
32 class PPPSRT:
33
34     msg_dic = {}
35     flag = b'\x7e'
36     addr = b'\xff'
37     ctrl_data = b'\x03'
38     ctrl_ack = b'\x07'
39     subs_esc = b'\x5d'
40     subs_flag = b'\x5e'
41     esc = b'\x7d'
42     max_payload_in_bytes = 1500
```

Figura 2: constantes.

Em seguida, algumas funções que serão recorrentes durante todo programa foram implementadas. A primeira delas foi a função `byte_stuffing(self, mse_in_bytes)`. Como se pode ver pelos argumentos, ela espera receber uma mensagem em bytes e não uma string comum. Esta começa criando uma string de bytes vazia chamada `msg_in_bytes_stuffed` e em um loop `for` percorre cada byte do argumento `msg_in_bytes`, colocando normalmente os bytes encontrados neste, naquele, exceto quando encontra um byte igual ao byte de *flag* ou de escape, ou seja, `0x7E` e `0x7D`, respectivamente. Neste caso, em vez de adicionar o mesmo byte em `msg_in_bytes_stuffed`, o byte de escape é adicionado primeiramente, seguido do byte modificado de *flag* ou de escape, como apresentado na figura 3.

É importante notar que na utilização de loop `for ... in ...` em uma string de bytes, os elementos são retratados como inteiros e não mais como bytes. Por conta disso, é preciso utilizar a função `chr()` para converter o inteiro em um caractere e logo depois a função `encode()` com a codificação `latin-1`, pois nela, diferentemente da codificação `utf-8`, todos os bytes possíveis podem ser convertidos para um par de números hexadecimais.

```

52 def byte_stuffing(self, msg_in_bytes):
53     msg_byte_stuffed = b''
54
55     for i in msg_in_bytes:
56
57         if (i == self.flag[0]):
58             msg_byte_stuffed = msg_byte_stuffed + self.esc + self.subs_flag
59             continue
60
61         if (i == self.esc[0]):
62             msg_byte_stuffed = msg_byte_stuffed + self.esc + self.subs_esc
63             continue
64
65         msg_byte_stuffed = msg_byte_stuffed + chr(i).encode('latin-1')
66
67     return msg_byte_stuffed

```

Figura 3: função de byte stuffing.

Depois desta função, a função `byte_destuffing(self, frame)` foi implementada. Esta, por sua vez, recebe um frame e retorna uma string em bytes. O processo descrito na última função é revertido, como mostra a figura 4.

```

71 def byte_destuffing(self, frame):
72     is_escape = False
73     new_frame = b''
74
75     for byte_int in frame:
76         byte = chr(byte_int).encode('latin-1')
77         if (is_escape):
78             byte_to_add = self.esc if (byte == self.subs_esc) else self.flag
79             new_frame = new_frame + byte_to_add
80             is_escape = False
81             continue
82         if (byte == self.esc):
83             is_escape = True
84             continue
85         new_frame = new_frame + byte
86
87     return new_frame

```

Figura 4: função de byte destuffing.

2.2 Funções de checksum

Para a função de *checksum* e para checá-lo, é inevitável criar uma função para somar os bytes do quadro até então. Para isso a função `sum_msg(self, msg)` foi criada. Ela começa tomando cada byte da mensagem e o transformando em sua representação binária. Em posse dessas representações, como o *checksum* usa 16 bits, os bytes são organizados da seguinte forma: o primeiro é posicionado nos 8 bits da direita. O próximo, nos 8 bits mais a esquerda. Estes dois formam uma palavra de 16 bits. Os próximos dois bytes são organizados da mesma forma, formando uma nova palavra. Estes dois são somados e armazenados na variável local `csm`. Daí em diante, cada palavra formada já é somada a esta

variável. No final, caso haja um número par de bytes todos eles serão somados e o valor final estará guardado em `csm`. Caso o número de bytes seja ímpar, então, ao se formar a última palavra, os 8 bits mais a esquerda serão todos zero. Também vale notar que no caso de uma soma resultar em 17 bits, com certeza o bit mais a esquerda será 1 e para retornar aos 16 bits desejados, este último bit mais a esquerda é eliminado e 1 é somado ao resto do número. Este processo pode ser visto no código da figura 5.

```

91     def sum_msg(self, msg):
92         csm = 0
93         elem = 0
94         aux = 'vazio'
95
96         for i in range(len(msg)):
97             if (i % 2 == 0):
98                 aux = f'{msg[i]:08b}'
99             else:
100                 if (i == 1):
101                     csm = int(f'{msg[i]:08b}' + aux, 2)
102                 else:
103                     elem = int(f'{msg[i]:08b}' + aux, 2)
104                     csm = elem + csm
105                     if (len(f'{csm:16b}') == 17):
106                         csm = csm + 1
107                         csm = int(f'{csm:16b}'[1:], 2)
108                 aux = ''
109
110         if (aux != ''):
111             elem = int(f'{0:08b}' + aux, 2)
112             csm = elem + csm
113             if (len(f'{csm:16b}') == 17):
114                 csm = csm + 1
115                 csm = int(f'{csm:16b}'[1:], 2)
116
117         return csm

```

Figura 5: função de somar os bytes da mensagem em palavras de 16 bits.

A função `checksum(self, msg_in_bytes)` simplesmente utiliza a função descrita anteriormente e calcula o seu complemento, retornando este complemento em bytes. Se a mensagem tiver um número ímpar de bytes, o *checksum* é convertido em bytes de forma *big endian*, senão, em *little endian*.

Já a função `check_checksum(self, frame)` recebe um quadro, tira os bytes de *flag* e calcula a soma com `sum_msg`. Caso essa soma resulte em 65535, ou seja, em 0xFFFF, a função retorna `True`, caso contrário, `False`. A figura 6 ilustra seu funcionamento e o da função anterior.

```

121     def checksum(self, msg_in_bytes):
122
123         csm = self.sum_msg(msg_in_bytes)
124
125         complement_csm = csm ^ 65535
126
127         mode = 'little' if (len(msg_in_bytes) % 2 == 0) else 'big'
128
129         return complement_csm.to_bytes(2, mode)
130
131
132     def check_checksum(self, frame):
133
134         frame_without_flags = frame[1:-1]
135
136         csm = self.sum_msg(frame_without_flags)
137
138         return csm == 65535
139

```

Figura 6: funções de checksum e sua checagem.

2.3 Função para enviar ACK

A função simples `send_ack(self, num)` apenas monta um quadro com as constantes conhecidas, com o campo de controle para ACK, e protocolo igual ao argumento `num`, mas em binário. Montado o quadro, ela o envia através do link que é iniciado em `send`.

```

143     def send_ack(self, num):
144         frame = self.addr \
145             + self.ctrl_ack \
146             + (num).to_bytes(2, 'big') \
147
148         frame = self.flag \
149             + frame \
150             + self.checksum(frame) \
151             + self.flag
152
153         frame = self.byte_stuffing(frame)
154
155         print(f'| sending stuffed ack: {frame}|')
156         self.link.send(frame)

```

Figura 7: função que envia uma mensagem de reconhecimento (ACK).

2.4 Função para extrair o payload de um quadro

A função `extract_msg(self, destuffed_frame)`, embora desnecessária, foi criada para uma melhor modularização. Ela apenas extrai o *payload* de um quadro.

```
161     def extract_msg(self, destuffed_frame):
162         msg = b''
163
164         for byte_int in destuffed_frame[5:-3]:
165             byte = byte_int.to_bytes(1, 'big')
166             msg = msg + byte
167
168         return msg
```

Figura 8: função para extrair o *payload* de um quadro.

2.5 Função send

A função `send(self, message)` começa com a criação de duas variáveis do tipo *array* e uma de bytes: `pieces_of_msg_in_bytes`, onde serão armazenadas as partes da mensagem a ser enviada, caso esta seja maior do que 1500 bytes, o tamanho máximo que o *payload* pode armazenar; `frames` onde os quadros para cada elemento da variável anterior serão armazenados; e a `msg_in_bytes`, que nada mais é do que a mensagem do argumento codificada em bytes com *latin-1* pelo motivo descrito no terceiro parágrafo da seção 2.1.

Em seguida, após a conversão da mensagem para um string de bytes, um loop é responsável por dividi-la em partes de até 1500 bytes, preenchendo assim a variável `pieces_of_msg_in_bytes`.

A partir daí, um outro loop é feito iterando cada uma das partes da mensagem guardadas nesta variável, e pra cada uma delas um quadro é criado. Primeiramente, o número de protocolo é calculado e codificado. Logo o quadro é criado apenas com os campos de endereço, controle, protocolo e *payload*. Isso é feito, pois assim é possível chamar a função `checksum` neste ponto sem ser necessário tirar partes do quadro. Em seguida, são adicionados a esse quadro os campos de *flag* inicial, o *checksum* e o de *flag* final. Neste ponto, o quadro está pronto para ser armazenado na variável `frames`.

Após o último loop, mais um loop é feito, dessa vez iterando cada um dos quadros da variável `frames`. Esta é a parte mais extensa da função `send`. No começo, uma variável booleana `acknowledged` com valor falso é iniciada, e outra variável inteira `stuffed_ack` que vai servir para guardar o ACK com o *byte stuffing*. Em seguida, o quadro da vez é enviado. O número de protocolo do quadro é guardado numa variável chamada `frame_protocol` e um contador é iniciado com zero.

Dai, um loop `while` é feito enquanto a variável `acknowledged` for negativa. Dentro deste loop é esperado receber a confirmação do envio do quadro. Assim, dentro de um `try`, o primeiro evento é o recebimento de uma mensagem de 12 bytes. Perceba que todos os ACKs serão de até este tamanho, pois o quadro terá 2 bytes de flag, 1 de endereço, 1 de controle, 2 de protocolo, 2 de *checksum* e de 2 a 4 bytes a mais após o *byte stuffing* que irá escapar os caracteres de *flag* e podem escapar caracteres no protocolo, caso este tenha 0x7D ou 0x7E. Perceba que estes são valores pequenos facilmente alcançáveis. O menor deles, 0x7D vale 125 em decimal. Se o recebimento deste ACK acontecer normalmente, em uma variável `ack` será guardada o quadro após o `byte_destuffing` e em outra variável `ack_protocol` o número de protocolo deste ACK será armazenado. Então, se o número de protocolo do ACK for o mesmo do do quadro, e se o *checksum* estiver correto, a variável `acknowledged` irá para `True`. No final o contador será incrementado e caso este tenha chegado ao número máximo de iterações, ou seja, se for maior que `max_tries` o loop será quebrado. Ainda no loop, após este bloco de `try`, há um bloco de `except TimeoutError` que será executado caso nada chegue no bloco `try`, que nada mais faz do que reenviar o quadro. A figura 9 explicita o que acabou de ser dito.

```

111         for frame in frames:
112             acknowledged = False
113             stuffed_ack = 0
114             self.link.send(frame)
115             frame_protocol = int.from_bytes(frame[4:6], 'big')
116             print(f'r sending {frame}, index: {frame_protocol}')
117             counter = 0
118             while (not acknowledged):
119                 try:
120                     stuffed_ack = self.link.recv(12)
121                     ack = self.byte_destuffing(stuffed_ack)
122                     ack_protocol = int.from_bytes(ack[3:5], "big")
123                     acknowledged = (ack_protocol == frame_protocol) and (self.check_checksum(ack))
124                     counter = counter + 1
125                     if (counter > self.max_tries):
126                         print(f'\n {frame} was NOT acknowledged!!!\n')
127                         break
128                     print(f' receiving ack: {ack} ---num--> {int.from_bytes(ack[3:5], "big")}')
129                     print(f' accepted: {self.check_checksum(ack)}')
130                 except TimeoutError:
131                     self.link.send(frame)
132             else:
133                 print(f'l frame {frame} was acknowledged!!!\n')

```

Figura 9: Parte principal da função `send`.

2.6 Função `recv`

A função `recv(self)` está envolta em um `try`, onde a sua exceção nada mais é do que um `exception TimeoutError` que imprime uma mensagem de erro.

Neste `try` o programa começa recebendo até 1500 bytes pelo *link*. O quadro sem o *byte stuffing* é armazenado na variável `pck`. Caso `pck` esteja vazio, a mensagem armazenada até então será exibida e o programa será finalizado. Caso contrário, o *checksum* será checado e no caso dele ser aceito a função

`self.send_ack(index)` será chamada, com `index` sendo o número do campo protocolo de `pck`. Por último, a mensagem é extraída do *payload* e armazenada no atributo `msg_dic` da classe. Esta mensagem extraída também é o que será retornado da função `recv`. A figura 10 mostra o seu funcionamento.

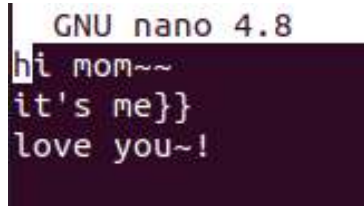
```
244     try:
245         stuffed_pck = self.link.recv(1500)
246         pck = self.byte_destuffing(stuffed_pck)
247
248         print(f'└ received package: {pck}')
249         if (pck == b''):
250             final_msg = ''
251             for i in self.msg_dic.keys():
252                 final_msg = final_msg + str(self.msg_dic[i], 'latin-1')
253             print(f'└ last package was b'' so the final message so far is:\n{final_msg}')
254             return b''
255
256         frame = pck
257         accept = self.check_checksum(frame)
258         index = int.from_bytes(frame[3:5], "big")
259         print(f'└ accept: {accept}')
260
261         if (accept):
262             self.send_ack(index)
263
264         msg = self.extract_msg(frame)
265         self.msg_dic[index] = msg
266
267         final_msg = msg
268
269     except TimeoutError:
270         print("Timeout")
271
272     print(f'└ final msg (return do recv): {final_msg}\n')
273     return final_msg
```

Figura 10: função `recv`.

3 Teste

3.1 Byte stuffing

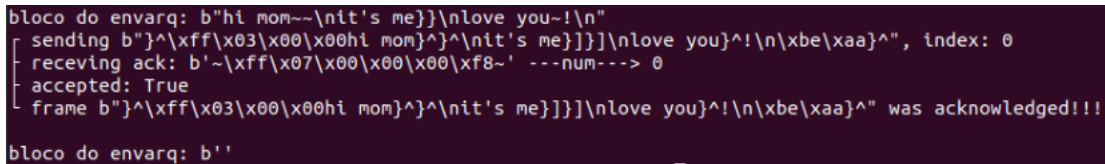
A primeira funcionalidade a ser testada foi a de *byte stuffing*. Para isso, no arquivo *send.txt* foi escrito a mensagem mostrada na figura 11.



```
GNU nano 4.8
hi mom~~
it's me}}
love you~!
```

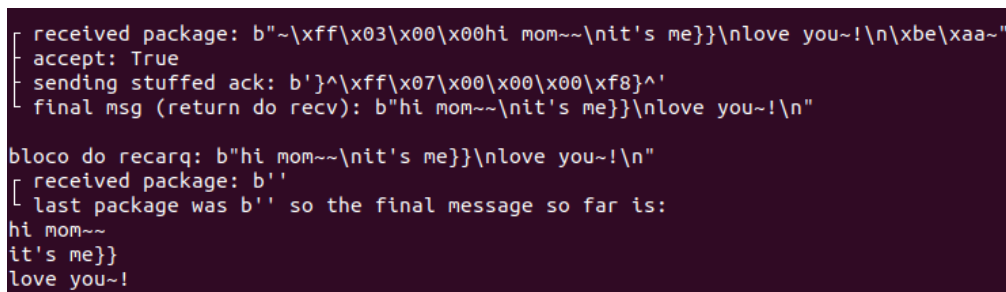
Figura 11: mensagem enviada.

Os caracteres `~` e `}` são codificados em *latin-1* para `0x7E` e `0x7D` respectivamente, por isso estão na mensagem. Vale salientar que os caracteres de hexadecimal `0x5E` e `0x5D` são, nesta ordem, `^` e `]`. Com base nisso, é possível notar que o *byte stuffing* está sendo executado corretamente na segunda linha da figura 12.



```
bloco do envarg: b"hi mom~~\nit's me}}\nlove you~!\n"
[ sending b"}^\xff\x03\x00\x00hi mom}^\nit's me}}]\nlove you}^\n\xbe\xaa}^", index: 0
[ receiving ack: b'~\xff\x07\x00\x00\x00\xf8~' ---num--> 0
[ accepted: True
[ frame b"}^\xff\x03\x00\x00hi mom}^\nit's me}}]\nlove you}^\n\xbe\xaa}^" was acknowledged!!!
bloco do envarg: b''
```

Figura 12: prompt após o a execução do programa *envarg.py*.



```
[ received package: b'~\xff\x03\x00\x00hi mom~~\nit's me}}\nlove you~!\n\xbe\xaa~"
[ accept: True
[ sending stuffed ack: b'}^\xff\x07\x00\x00\x00\xf8}^'
[ final msg (return do recv): b"hi mom~~\nit's me}}\nlove you~!\n"

bloco do recarg: b"hi mom~~\nit's me}}\nlove you~!\n"
[ received package: b''
[ last package was b'' so the final message so far is:
hi mom~~
it's me}}
love you~!
```

Figura 13: prompt após o a execução do programa *recarg.py*.

No lado do *recarg.py* o quadro está passando pela função `byte_destuffing` também corretamente, como mostra a primeira linha da figura 13. Além disso, o *checksum* está sendo aceito, pois está correto. Deste ponto, a mensagem de ACK é enviada de volta para o *envarg.py* e a mensagem recebida é mostrada na

tela. Como o próximo quadro recebido está vazio, a mensagem final é mostrada na tela.

De volta ao lado do *envarq.py*, após o recebimento do ACK, o seu número de protocolo é mostrado na tela, assim como o resultado da checagem do *checksum*.

3.2 Checksum

Outro ponto que deve ser checado é se a mensagem é descartada caso o *checksum* esteja errado. Para isso, apenas temporariamente, o tamanho máximo do *payload* foi modificado de 1500 bytes para apenas 10, pois assim 3 quadros serão enviados, permitindo uma melhor manipulação do processo.

```
if (i != 2):
    frame = self.flag \
        + frame \
        + self.checksum(frame) \
        + self.flag
else:
    frame = self.flag \
        + frame \
        + b'\x00\x00' \
        + self.flag
```

Figura 14: Modificação na montagem do quadro para testar o *checksum*.

Também foram incluídas mudanças para enviar um *checksum* errado no quadro de índice 1, só enviado o correto no reenvio. Na primeira vez, o *checksum* é setado para zero (figura 14). Já no reenvio ele é calculado corretamente (figura 15).

```
except TimeoutError:
    if (frame_protocol == 1):
        print(f'| resending...')
        new_frame = self.addr \
            + self.ctrl_data \
            + (1).to_bytes(2, 'big') \
            + pieces_of_msg_in_bytes[1]

        new_frame = self.flag \
            + new_frame \
            + self.checksum(new_frame) \
            + self.flag
        self.link.send(self.byte_stuffing(new_frame))
    else:
        self.link.send(frame)
```

Figura 15: Modificação para correto envio do *checksum* na etapa de reenvio.

Como se pode ver nas figura 16 e 17, o reenvio acaba ocorrendo, pois o *checksum* não está tendo o resultado esperado.

```

bloco do envarg: b'hi mom~~\nit's me}}\nlove you~!\n"
[ sending b'}^\xff\x03\x00\x00hi mom}^\^ni\x7f\xd0}^', index: 0
[ receiving ack: b'~\xff\x07\x00\x00\x00\xf8~' ---index---> 0
[ accepted: True
[ frame b'}^\xff\x03\x00\x00hi mom}^\^ni\x7f\xd0}^' was acknowledged!!!

[ sending b"}^\xff\x03\x00\x01t's me}}]\nl\x00\x00}^", index: 1
[ resending...
[ receiving ack: b'~\xff\x07\x00\x01\x00\xf7~' ---index---> 1
[ accepted: True
[ frame b"}^\xff\x03\x00\x01t's me}}]\nl\x00\x00}^" was acknowledged!!!

[ sending b'}^\xff\x03\x00\x02ove you}^!\n\x1ck}^', index: 2
[ receiving ack: b'~\xff\x07\x00\x02\x00\xf6~' ---index---> 2
[ accepted: True
[ frame b'}^\xff\x03\x00\x02ove you}^!\n\x1ck}^' was acknowledged!!!

bloco do envarg: b''

```

Figura 16: prompt no teste do *checksum* no arquivo *envarg.py*.

```

[ received package: b'~\xff\x03\x00\x00hi mom~~\ni\x7f\xd0~'
[ accept: True
[ sending stuffed ack: b'}^\xff\x07\x00\x00\x00\xf8}^'
[ final msg (return do recv): b'hi mom~~\ni'

bloco do recarg: b'hi mom~~\ni'
[ received package: b"~\xff\x03\x00\x01t's me}}\nl\x00\x00~"
[ accept: False
[ final msg (return do recv): b"t's me}}\nl"

bloco do recarg: b"t's me}}\nl"
[ received package: b"~\xff\x03\x00\x01t's me}}\nl$~"
[ accept: True
[ sending stuffed ack: b'}^\xff\x07\x00\x01\x00\xf7}^'
[ final msg (return do recv): b"t's me}}\nl"

bloco do recarg: b"t's me}}\nl"
[ received package: b'~\xff\x03\x00\x02ove you~!\n\x1ck~'
[ accept: True
[ sending stuffed ack: b'}^\xff\x07\x00\x02\x00\xf6}^'
[ final msg (return do recv): b'ove you~!\n'

bloco do recarg: b'ove you~!\n'
[ received package: b''
[ last package was b'' so the final message so far is:
hi mom~~
it's me}}
love you~!

bloco do recarg: b''

```

Figura 17: prompt no teste do *checksum* no arquivo *recarg.py*.

3.3 ACK

No caso do ACK, é necessário verificar se o reenvio ocorre quando o ACK é perdido ou está errado. O caso da falta de ACK já foi verificado na seção

anterior, pois ao detectar que o *checksum* está errado o *recarq.py* não envia o ACK, o que implica no reenvio do quadro pelo *envarq.py*.

Quanto ao erro no ACK, ele pode acontecer de duas formas: erro no *checksum* e erro no índice. O primeiro consiste de um erro tratado na seção anterior. O segundo será demonstrado aqui.

Para isso, as bibliotecas *math* e a função *random()* da biblioteca *random* foram importadas. A modificação feita foi apenas uma mudança aleatória no índice do ACK enviado quando este deveria ser 1. Ou seja, quando o quadro 1 for recebido, o será gerado um número aleatório entre 0 e 4 para ser o índice do ACK enviado. Assim, o ACK só deve ser aceito quando o índice for gerado aleatoriamente. As mudanças podem ser vistas nas figuras 18 e 19, bem como o seu resultado nas figuras 20 e 21.

```
33 import math
34 from random import random
```

Figura 18: bibliotecas importadas.

```
257         if (accept):
258             if (index == 1):
259                 new_index = math.floor(random()*5)
260                 print(f'new index = {new_index}')
261                 self.send_ack(new_index)
262             else:
263                 self.send_ack(index)
```

Figura 19: modificação na função *recv* para gerar um índice aleatório para o ACK de índice 1.

```
[ sending b'~\xff\x03\x00\x00hi mom}^^\n\x7f\xd0}^', index: 0
+ receiving ack: b'~\xff\x07\x00\x00\x00\xf8~' ---index--> 0
+ accepted: True
+ frame b'~\xff\x03\x00\x00hi mom}^^\n\x7f\xd0}^' was acknowledged!!!

[ sending b'~\xff\x03\x00\x00it's me}}]\n\x5d}^', index: 1
+ receiving ack: b'~\xff\x07\x00\x03\x00\xf5~' ---index--> 3
+ accepted: True
+ receiving ack: b'~\xff\x07\x00\x00\x00\xf8~' ---index--> 0
+ accepted: True
+ receiving ack: b'~\xff\x07\x00\x01\x00\xf7~' ---index--> 1
+ accepted: True
+ frame b'~\xff\x03\x00\x00it's me}}]\n\x5d}^' was acknowledged!!!

[ sending b'~\xff\x03\x00\x00ove you}^!\n\x1ck}^', index: 2
+ receiving ack: b'~\xff\x07\x00\x02\x00\xf6~' ---index--> 2
+ accepted: True
+ frame b'~\xff\x03\x00\x00ove you}^!\n\x1ck}^' was acknowledged!!!
```

Figura 20: prompt no teste do índice do ACK no arquivo *envarq.py*.


```

[ received package: b'~\xff\x03\x00\x00hi mom~~\n\x7f\xd0~'
[ accept: True
[ sending stuffed ack: b'}^\xff\x07\x00\x00\x00\xf8}^'
[ final msg (return do recv): b'hi mom~~\n'

[ received package: b"~\xff\x03\x00\x01t's me}}\n\x5d~"
[ accept: True
new index = 3
[ sending stuffed ack: b'}^\xff\x07\x00\x03\x00\xf5}^'
[ final msg (return do recv): b"t's me}}\n"

[ received package: b"~\xff\x03\x00\x01t's me}}\n\x5d~"
[ accept: True
new index = 0
[ sending stuffed ack: b'}^\xff\x07\x00\x00\x00\xf8}^'
[ final msg (return do recv): b"t's me}}\n"

[ received package: b"~\xff\x03\x00\x01t's me}}\n\x5d~"
[ accept: True
new index = 1
[ sending stuffed ack: b'}^\xff\x07\x00\x01\x00\xf7}^'
[ final msg (return do recv): b"t's me}}\n"

[ received package: b'~\xff\x03\x00\x02ove you~!\n\x1ck~'
[ accept: True
[ sending stuffed ack: b'}^\xff\x07\x00\x02\x00\xf6}^'
[ final msg (return do recv): b'ove you~!\n'

[ received package: b''
[ last package was b'' so the final message so far is:
hi mom~~
it's me}}
love you~!

```

Figura 21: prompt no test do índice do ACK no arquivo *recarq.py*.

Perceba que ao chegar no ACK de índice 1, dois índices aleatórios foram enviados antes de ser enviado o índice correto. O primeiro índice foi 3 e o segundo foi 0. Perceba que no *envarq.py*, ambos foram aceitos, pois tinham *checksum* corretos, porém, como não tinham os índices corretos, eles não foram usados para validar a entrega do quadro que havia sido enviado. Apenas quando o ACK de índice correto foi enviado, isto é, o ACK de índice 1, o *envarq.py* aceitou e continuou com a entrega dos outros quadros.

Vale salientar que esta "confusão" com os índices do ACK e o reenvio do mesmo quadro diversas vezes não modificou a mensagem final exibida em *recarq.py*. Ou seja, a mensagem do quadro em questão ("t's me}}") não foi repetida. Isso porque o programa está feito para sempre sobrescrever a parte da mensagem que tiver o índice repetido. Por exemplo, se chegar um quadro de

índice 3 com o *payload* "casa", completando a mensagem "eu tenho uma casa" e logo em seguida chegar outro quadro de índice 3, mas dessa vez com o *payload* "mansão", caso a transmissão se finalize, a mensagem final será "eu tenho uma mansão". Como no problema o *payload* do quadro é sempre igual, isso não é um problema.

4 Conclusão

Como se pode ver, o código feito em *pppsrt.py* está preparado para os erros mais comuns, tanto o erro de *checksum* nas mensagens normais e nos ACKs, quanto nos erros no índice dos ACKs. As comunicações estão acontecendo normalmente. Todos os campos estão presentes e cumprindo suas funções. O único teste não feito foi o de recuperação de mensagens com *flags* no meio do quadro, o que pode atrapalhar todo o resto da transmissão desse ponto em diante.

O código segue em anexo no arquivo zip.