



Universidade de Brasília
Departamento de Ciências da Computação
Engenharia de Computação
Teleinformática e Redes II

Selective Repeat

João Pedro Gomes Covalleski Marin Antonow, 221006351

Pedro Amorim de Gregori, 221029329

Álvaro Luz, 180115391

Professor: Jacir Luiz Bordim

Brasília
2023

1. Objetivos

1. Trabalhar com uma arquitetura de rede em camadas e implementar funções que permitam uma melhor utilização do enlace utilizando técnicas de pipelining vistas no capítulo 4 (Camada de Transporte) do livro.
2. Seu trabalho é implementar o rdt_4_0, estendendo/modificando o código inicial fornecido (stop-and-wait) para que múltiplos pacotes possam fluir entre cliente e servidor.
3. Implementações possíveis:
 - (a) Go-back-N (mais simples)
 - (b) Selective Repeat (mais complexo, e por isso tem um bônus extra na nota final para o grupo que implementar de forma correta).
4. Após a execução/simulação, seu código deve fornecer as seguintes estatísticas:
 - (a) Vazão (camada de rede, incluindo cabeçalhos)
 - (b) Goodput (vazão na camada de aplicação)
 - (c) Total de pacotes transmitidos
 - (d) Total de retransmissões (para cada tipo de pacote utilizado)
 - (e) Total de pacotes corrompidos (para cada tipo de pacote utilizado)
 - (f) Tempo de simulação (tempo desde o início do envio até o último pacote enviado)
5. O código deve permitir o envio de múltiplas mensagens entre o cliente e servidor. O número de mensagens deve ser definido como argumento de linha do cliente.

2. Projeto

A nossa implementação consistiu em modificar o código-base proposto e, além de implementar o mecanismo de selective repeat para a transmissão de dados, elaborar um gráfico automático através dos recursos da biblioteca *matplotlib*, a fim de obter uma visualização pacote-por-pacote do throughput e do goodput referentes à transmissão.

3. Código

3.1. Cliente

Basicamente, o funcionamento do cliente consiste, previamente ao envio, em expandir o quantitativo de mensagens originais de acordo com o argumento de linha fornecido pelo usuário, resultando em um total de $(n \cdot 5) + 5$ mensagens.

```
#0 código deve permitir o envio de multiplas mensagens entre o cliente e servidor.
# 0 número de mensagens deve ser definido como argumento de linha do cliente.
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Quotation client talking to a Pig Latin server.')
    parser.add_argument('server', help='Server.')
    parser.add_argument('port', help='Port.', type=int)
    parser.add_argument('num_msg', help='Number of Messages.', type=int)
    args = parser.parse_args()
    messages = []

    msg_L = [
        'Microsoft is not evil, they just make really crappy operating systems - Linus Torvalds',
        'Real programmers can write assembly code in any language - Larry Wall',
        'It is hardware that makes a machine fast. It is software that makes a fast machine slow. -- Craig Bruce',
        'The art of debugging is figuring out what you really told your program to do rather than what you thought you told it to do. -- Andrew Singer',
        'The computer was born to solve problems that did not exist before. - Bill Gates']

    msg_L_aux = msg_L[:]
    totmsgbytes = 0

    #Aumentar numero de mensagens de acordo com o argumento de linha - total de msgs = n*5 + 5
    for _ in range(args.num_msg):
        msg_L.extend(msg_L_aux)

    timeout = 1000 # send the next message if not response
    rdt = RDT.RDT('client', args.server, args.port)
    in_order = {}
```

Figura 1. Cliente - Pré-envio

O evento posterior consiste em passar essas mensagens para a camada de transporte e enviá-las ao servidor, através do método `rdt_4_0_send`, passando como argumento a lista que contém todas as mensagens.

```
try:
    begin = time.time()
    #Printa todas as mensagens que vao ser enviadas para o servidor
    for message in msg_L:
        print('Client asking to change case: ' + message)

    # try to receive message before timeout
    rdt.rdt_4_0_send(msg_L)
    rdt.clear() #limpa as variaveis e afins do rdt

    time_of_last_data = time.time()
    send_time = time_of_last_data - begin
```

Figura 2. Cliente - Envio de Mensagens

Após a confirmação de que todas as mensagens foram recebidas pelo servidor, o cliente para de transmitir e passa a estar pronto para receber as mensagens convertidas em *Caps Lock* e armazená-las em uma lista, até que o caractere especial terminador de transmissão, `\0`, é recebido. Quando o cliente receber esse caractere, ele irá estabelecer um período de 2[s] para garantir que mais nenhuma mensagem seja recebida e, caso haja o recebimento, reenviar-se-á o ack referente à mensagem recebida. Após o timeout, o cliente organizará as mensagens e apresentará as estatísticas referentes à transmissão das mensagens.

```
print("Client: receiving messages")

while True:
    msg_S = None
    msg_seq = None
    while msg_S == None:
        (msg_seq, msg_S) = rdt.rdt_4_0_receive() #recebimento mensagem por mensagem
        if msg_S is None:
            if time_of_last_data + timeout < time.time():
                break
            else:
                continue
    time_of_last_data = time.time()

    # caso de receber o caractere especial para parar o recebimento de strings
    # crucial para o servidor parar de enviar e para o cliente saber quantas mensagens ha no total
    if(msg_S == "\0"):
        print("\nClient: received special message to stop receiving")
        timer = time.time() #tempo extra para garantir que o ack sera corretamente enviado para o servidor
        while (timer+2 > time.time()):
            (seq_L, msg_L) = rdt.rdt_4_0_receive()
            break

    if msg_seq not in in_order: #adiciona as mensagens recebidas fora de ordem
        in_order[msg_seq] = msg_S
```

Figura 3. Cliente - Recebimento de Mensagens

```

pkts = sum(rdt.network.pktsent) #soma de todos os bytes dos pacotes enviados

avg_throughput = pkts/send_time #average throughput

gpkts = sum(rdt.goodput) #soma de todos os bytes de dados dos pacotes enviados

avg_goodput = gpkts/send_time #average goodput

msg_convertidas = rdt.reorder(in_order) #reordenacao no sistema final das mensagens recebidas

#print(msg_convertidas)
for msg_S in msg_convertidas:
    print('\nClient: Received the converted frase to: ' + msg_S + '\n')

#overview das estatisticas
debug_stats(f"Simulation time = {(time.time()-begin):.2f}[s]")
debug_stats(f"Throughput = {avg_throughput:.2f}[Bps]")
debug_stats(f"Goodput = {avg_goodput:.2f}[Bps]")
debug_stats(f"Total of packets in the wire (ack+data+end) = {rdt.totalpackets+rdt.totalacks}")
debug_stats(f"Total of transmited packets = {rdt.totalpackets}")
debug_stats(f"Total of data packets = {rdt.totaldata}")
debug_stats(f"Total of ack packets = {rdt.totalacks}")
debug_stats(f"Total of end char needed = {rdt.endchar}")
debug_stats(f"Total of lost packets (data + ack) = {rdt.totallostpkts}")
debug_stats(f"Total of corrupted acks = {rdt.totalcorrupted_acks}")
debug_stats(f"Total of corrupted packets = {rdt.totalcorrupted}")
debug_stats(f"Total of reordered packets = {rdt.totalreordered}")
debug_stats(f"Total of retransmitted packets = {rdt.totalretransmitted}")

```

Figura 4. Cliente - Reordenação e Estatísticas

Há a disponibilização de gráficos adicionais de throughput e goodput por pacote, ou seja, o fornecimento da visualização do throughput e goodput de cada pacote transmitido.

```

#graficos do throughput por pacote enviado
pktsent = rdt.network.pktsent
timelist = rdt.network.timerlist
throughput = [(a / b)/1e3 for a, b in zip(pktsent,timelist)]
fig, (a1,a2) = plt.subplots(2,1)

plt.subplots_adjust(hspace=0.8)

a1.grid(True)
a1.scatter(timelist, throughput, c='red', edgecolors='black', linewidths=1,alpha=0.75)
for pkth, time in zip(throughput, timelist):
    a1.annotate('',xy=(time,pkth), xytext= (10,-10), textcoords='offset points')
a1.set_title("Throughput X Time - Client")
a1.set_ylabel("Throughput [kB/s]")
a1.set_xlabel("Time [s]")

#graficos do goodput por pacote enviado
a2.grid(True)
pkggoodput = rdt.goodput
timelist_goodput = rdt.timerlist
goodput = [(a/b)/1e3 for a,b in zip(pkggoodput,timelist_goodput)]

a2.scatter(timelist_goodput, goodput, c='red', edgecolors='black', linewidths=1,alpha=0.75)
for pkg, time2 in zip(goodput, timelist_goodput):
    a2.annotate('',xy=(time2,pkg), xytext= (10,-10), textcoords='offset points')
a2.set_title("Goodput X Time - Client")
a2.set_ylabel("Goodput [kB/s]")
#a2.set_yscale('log')
a2.set_xlabel("Time [s]")
plt.show()

```

Figura 5. Cliente - Gráficos de throughput e goodput por pacote

3.2. Servidor

O servidor, por sua vez, tem funcionamento análogo ao do cliente, mas papéis trocados: inicialmente, ele está programando para receber infinitas mensagens, pois não sabe quantas mensagens o cliente estará enviando. A cada mensagem recebida, esta se submeterá à função *upperCase* e será convertida em *Caps Lock*, além de ser armazenada em uma lista. No entanto, caso haja o recebimento do caractere especial `\0`, o servidor fará o mesmo procedimento descrito anteriormente: aguardar 2[s] para garantir que não haja o recebimento de nenhuma mensagem e, dado o timeout, parar de receber mensagens e começar a enviar as mensagens convertidas.

```
try:
    begin = time.time()
    while True:
        # try to receiver message before timeout
        time_of_last_data = time.time()
        (seq_L,msg_L) = rdt.rdt_4_0_receive() #recebimento de mensagens
        if msg_L is None:
            if time_of_last_data + timeout < time.time():
                break
            else:
                continue
        time_of_last_data = time.time()

        # caso de receber o caractere especial para parar o recebimento de strings
        # crucial para o cliente parar de enviar e para o servidor saber quantas mensagens ha no total
        if(msg_L=="\0"):
            print("\nServer: received special message to stop converting")
            timer = time.time() #tempo extra para garantir que o ack sera corretamente enviado para o servidor
            while (timer+2 > time.time()):
                (seq_L,msg_L) = rdt.rdt_4_0_receive()
            break

        # convert and reply
        if(seq_L not in send_in_order): #adiciona as mensagens recebidas fora de ordem e converte em CAPS LOCK
            rep_msg_L = upperCase(msg_L)
            send_in_order[seq_L] = rep_msg_L
            print(f"\nmsgs_rcvs == {send_in_order}")
            print('\nServer: converted %s \nto %s\n' % (msg_L, rep_msg_L))
```

Figura 6. Servidor - Recebimento

O evento posterior consiste em passar essas mensagens convertidas para a camada de transporte e enviá-las ao cliente, através do método `rdt_4_0_send`, passando como argumento a lista que contém todas as mensagens convertidas. Assim como no cliente, serão disponibilizadas as estatísticas de envio de mensagens pelo lado servidor, baseando-se nas mesmas métricas adotadas no outro lado da comunicação.

```

#reordenacao no sistema final a conversao em caps lock das mensagens recebidas
server_rcv = rdt.reorder(send_in_order)
print("\nServer: sending converted messages")
rdt.clear() #limpeza das variaveis
begin = time.time()
rdt.rdt_4_0_send(server_rcv) #envio das mensagens convertidas
send_time = time.time() - begin #tempo de envio

pkts = sum(rdt.network.pktsent) #soma de todos os bytes dos pacotes enviados
avg_throughput = pkts/send_time #average throughput

gpkts = sum(rdt.goodput) #soma de todos os bytes de dados dos pacotes enviados
avg_goodput = gpkts/send_time #average goodput

#overview das estatisticas
debug_stats(f"Simulation time = {(time.time()-begin):.2f}[s]")
debug_stats(f"Throughput = {avg_throughput:.2f}[Bps]")
debug_stats(f"Goodput = {avg_goodput:.2f}[Bps]")
debug_stats(f"Total of packets in the wire (ack+data+end) = {rdt.totalpackets+rdt.totalacks}")
debug_stats(f"Total of transmited packets = {rdt.totalpackets}")
debug_stats(f"Total of data packets = {rdt.totaldata}")
debug_stats(f"Total of ack packets = {rdt.totalacks}")
debug_stats(f"Total of end char needed = {rdt.endchar}")
debug_stats(f"Total of lost packets (data + ack) = {rdt.totallostpkts}")
debug_stats(f"Total of corrupted acks = {rdt.totalcorrupted_acks}")
debug_stats(f"Total of corrupted packets = {rdt.totalcorrupted}")
debug_stats(f"Total of reordered packets = {rdt.totalreordered}")
debug_stats(f"Total of retransmitted packets = {rdt.totalretransmitted}")

```

Figura 7. Servidor - Envio de Mensagens e overview das estatísticas

Assim como no lado do cliente, serão disponibilizados o gráfico de throughput e goodput por pacote transmitido referente ao envio do servidor.

```

#graficos do throughput por pacote enviado
pktsent = rdt.network.pktsent
timelist = rdt.network.timerlist
throughput = [(a / b)/1e3 for a, b in zip(pktsent,timelist)]
fig, (a1,a2) = plt.subplots(2,1)

plt.subplots_adjust(hspace=0.8)

a1.grid(True)
a1.scatter(timelist, throughput, c='red', edgecolors='black', linewidths=1,alpha=0.75)
for pktth, time in zip(throughput, timelist):
    a1.annotate('',xy=(time,pktth), xytext= (10,-10), textcoords='offset points')
a1.set_title("Throughput X Time - Server")
a1.set_ylabel("Throughput [kB/s]")
a1.set_xlabel("Time [s]")

#graficos do goodput por pacote enviado
a2.grid(True)
pkgoodput = rdt.goodput
timelist_goodput = rdt.timerlist
goodput = [(a/b)/1e3 for a,b in zip(pkgoodput,timelist_goodput)]

a2.scatter(timelist_goodput, goodput, c='red', edgecolors='black', linewidths=1,alpha=0.75)
for pkg, time2 in zip(goodput, timelist_goodput):
    a2.annotate('',xy=(time2,pkg), xytext= (10,-10), textcoords='offset points')
a2.set_title("Goodput X Time - Server")
a2.set_ylabel("Goodput [kB/s]")
a2.set_xlabel("Time [s]")
plt.show()

```

Figura 8. Servidor - Gráficos de throughput e goodput por pacote

3.3. RDT

O RDT foi construído baseado em 2 métodos principais: `rdt_4_0_send` e `rdt_4_0_receiver`. O primeiro método recebe um conjunto de mensagens, transforma-os,

um-por-um, em pacotes e aplica a técnica *selective repeat* para melhor controle e performance do enlace para os pacotes enviados. O segundo método, por sua vez, tem por objetivo verificar o recebimento dos pacotes, avaliar o estado deles e enviar uma resposta que condiz com o estado verificado.

3.3.1. Observações

Para o desenvolvimento do trabalho, consideramos os seguintes códigos referentes às respostas advindas da camada de rede:

1. Um ACK de um pacote tem que ser uma string correspondente ao seu número de sequência. Ex: numSeq=0 recebe ack=0, e assim sucessivamente.
2. Um pacote corrompido vai ter número de sequência aleatório e sua mensagem será "N", similar ao NACK.


3.3.2. Sender

O `rdt_4_0_send` foi implementado baseado na técnica *selective repeat*; desse modo, foi necessária a criação de um algoritmo que aplicasse uma janela deslizante para controlar o envio das mensagens.

Inicialmente, há a transformação de todas as mensagens em pacotes, que são adicionados a uma lista *packets*. Além disso, cria-se um dicionário vazio nomeado *pack_ack*, o qual possuirá suas chaves correspondentes ao número de sequência do pacote e seus valores correspondentes ao ACK recebido.

O tamanho da janela foi fixado em 5 pacotes e ela desliza somente quando o pacote de menor número de sequência recebe um ACK; por consequência e aplicação do SR, a próxima base da janela será o pacote com menor número de sequência que ainda não recebeu o ACK. Seguindo as fronteiras da janela, o código envia os pacotes e esperam por um ACK, no entanto, quando ocorre um evento de TIMEOUT no pacote, recebimento de NACK (Pacote ou Ack Corrompido) ou quando se verifica um erro de reordenamento dos pacotes, o programa reenvia o pacote que sofreu alguma interferência. Estas interferências estão sujeitas a mudanças de acordo com as probabilidades de atributos da classe Network, em que uma instância da mesma é atributo do RDT o qual estudamos.

O algoritmo foi implementado em um loop principal que verifica se todas as mensagens foram enviadas. Em caso negativo, ocorre uma iteração apenas através da janela deslizante, realizando os envios, as verificações de timeout e o recebimento das respostas.



```
1 while(len(pack_ack) != len(packets)):
2     for packet in packets[lowest_seq : lowest_seq + self.window_size]:
```

Figura 9. Loop principal

Se um pacote recebeu um ACK anteriormente e há o pacote e o ACK em *pack_ack*, a iteração prossegue, pois não se faz necessário o reenvio, conforme a técnica SR.

```
1 if(packet.seq_num in pack_ack):
2     #selective repeate nao retransmite se ja recebi o ack
3     continue
```

Figura 10. Verifica se precisa enviar

Para a contagem da métrica de retransmissão e transmissão de pacotes, foi adicionado o código abaixo.

```
1 if(packet.seq_num in transmited):
2     self.totalretransmited += 1 #ja retransmiti o mesmo pacote
3 else:
4     transmited.append(packet.seq_num) #primeira vez transmitindo
```

Figura 11. Verifica se precisa enviar

O envio efetivo de mensagens se dá pela aplicação do método de envio de objeto da camada de rede, atributo da camada de transporte. Caso não haja resposta, computa-se um pacote ou ack perdido e continua a iteração. Caso haja resposta, segue-se no método.

```
self.network.udt_send(packet.get_byte_S())
response = ''
timer = time.time()

while response == '' and (timer + self.timeout > time.time()):
    response = self.network.udt_receive()

send_time = time.time() - timer #tempo de envio por pacote

#metricas pro calculo do throughput
self.network.timerlist.append(send_time)
self.network.pktsent.append(throughput_byte)

if response == '':
    #ack ou pkt nao recebido no receiver
    debug_log("SENDER: Packet Lost")
    self.totallostpkts += 1
    continue

debug_log("SENDER: " + response)
msg_length = int(response[:Packet.length_S_length])
self.byte_buffer = response[msg_length:]

self.totalpackets += 1 #pacotes com resposta
```

Figura 12. Envio Efetivo

Em caso de qualquer resposta e posteriormente à aplicação das métricas, tem-se a verificação da resposta, que se dá através de um conjunto de condicionais, começando pela verificação se a resposta está corrompida:

```
1 if not Packet.corrupt(response[:msg_length]):
2     #pacote nao foi corrompido
3     response_p = Packet.from_byte_S(response[:msg_length])
4     debug_log(response_p.msg_S)
```

Figura 13. Response not Corrupt

Quando recebido um NACK, o programa incrementa a contagem de pacotes corrompidos na transmissão, o que significa que o receptor do outro lado da comunicação recebeu um pacote corrompido.

```
1 if (response_p.msg_S == "N"):
2     #pacote corrompido no receiver
3     debug_log("SENDER: PACKET CORRUPTED")
4     self.byte_buffer = ''
5     self.totalcorrupted += 1
```

Figura 14. Pacote Corrompido

No caso de a resposta já estar no dicionário que computa os pacotes que receberam os ACKS, considera-se que o receptor está atrasado em relação ao emissor e, portanto, há erro de reordenamento na camada de rede do RDT. Como orientado pelo professor, calcula-se, ainda assim, o goodput, já que há cabeçalho de dados em questão.

```
elif response_p.seq_num in pack_ack:
    #resposta de pacote que ja tem ack
    if (pack_ack[response_p.seq_num] == f"{response_p.msg_S}"):
        debug_log("SENDER: Receiver behind sender, probably reordered")
        self.totalreordered += 1
        test = Packet(response_p.seq_num, f"{packet.seq_num}")
        self.network.udt_send(test.get_byte_S())
        self.goodput_bytes += goodput_byte
        self.goodput.append(goodput_byte)
        self.timerlist.append(send_time)
```

Figura 15. Reordenamento - Caso 1

Quando recebido um ACK, adiciona-se a chave (número de sequência do pacote transmitido) e o valor (ACK) ao dicionário *pack_ack*. Além disso, programa calcula se a janela precisará ser deslocada, a posição para a qual ela deverá ou não ir e as métricas de goodput.

```

1 elif (response_p.msg_S == f"{packet.seq_num}"):
2     debug_log("NEW PACKET")
3     debug_log("SENDER: ACK received")
4
5     pack_ack[packet.seq_num] = response_p.msg_S
6
7     self.totalacks += 1
8     self.totaldata += 1
9
10    #metricas para calculo do goodput
11    self.goodput_bytes += goodput_byte
12    self.goodput.append(goodput_byte)
13    self.timerlist.append(send_time)
14
15    self.send_time += send_time

```

Figura 16. ACK recebido

```

1 if response_p.seq_num == packets[lowest_seq].seq_num:
2     for key in packets:
3         if key.seq_num not in pack_ack:
4             lowest_seq = key.seq_num
5             break

```

Figura 17. Controle da janela

No caso do pacote e o ACK não terem sido corrompidos, a resposta não conferir com o número de sequência do pacote transmitido e for uma resposta diferente das respostas já recebidas e confirmadas com ACK, também será considerado como um caso de reordenamento.

```

else:
    #se o pacote nao foi corrompido, o ack nao foi corrompido,
    #nao eh o ack do pacote e nao foi recebido ainda
    debug_log("SENDER: Receiver behind sender, probably reordered")
    self.totalreordered += 1
    test = Packet(response_p.seq_num, f"{packet.seq_num}")
    self.network.udt_send(test.get_byte_S())
    self.goodput_bytes += goodput_byte
    self.goodput.append(goodput_byte)
    self.timerlist.append(send_time)

self.byte_buffer = ''

```

Figura 18. Reordenamento - Caso 2

Para a sinalização do fim do envio das mensagens, o Sender envia um pacote com um caractere especial e espera obter alguma resposta do Receiver, seguindo o algoritmo acima, excetuando-se a parte da janela, pois só existe um pacote a ser enviado no momento. Desconsiderar-se-á o efeito do envio desse caractere para o cálculo da vazão e do goodput, dado que seu tamanho destoa das demais amostras e leva a resultados fora do esperado.

```
#envio do caractere especial para parar a conversao no receiver
while True:
    packet = Packet(999999999, "\0")
    self.network.udt_send(packet.get_byte_S())
    response = ''

    self.totalpackets += 1
    self.endchar += 1

    debug_log(f"SENDER: TRANSMITING PACKET - END CHAR -> {packet.msg_S}")

    timer = time.time()

    while response == '' and (timer + self.timeout > time.time()):
        response = self.network.udt_receive()

    send_time = time.time() - timer

    if response == '':
        debug_log("SENDER: 'End Char' Packet Lost")
        self.totallostpkts += 1
        continue

    msg_length = int(response[:Packet.length_S_length])
    self.byte_buffer = response[msg_length:]
```

```
if not Packet.corrupt(response[:msg_length]):
    response_p = Packet.from_byte_S(response[:msg_length])

    if (response_p.msg_S == "\0"):
        debug_log("SENDER: ACK RECEIVED")
        self.send_time += send_time
        break

    else:
        self.totalcorrupted += 1
        self.byte_buffer = ''

    self.byte_buffer = ''
else:
    self.totalcorrupted_acks += 1
    continue
```

Figura 19. End Char

Uma vez recebido o ACK referente ao caractere especial, finda-se a execução do método.

3.3.3. Receiver

O método `rdt.4.0_receiver` foi implementado para a recepção das mensagens, bufferização e envio de ACKS para o RDТ Sender do outro lado da aplicação, funci-

onando de forma a verificar o recebimento das mensagens e elaborar uma resposta de acordo com os pacotes que recebe. Sendo assim, ela fica encarregada de receber as mensagens, enviar ACKs quando recebe pacotes em bom estado e enviar NACKs quando o pacote esta corrompido. A imagem abaixo mostra os passos iniciais do RDT receiver, que consistem em capturar uma mensagem advinda da camada de rede e inicializar um *pack_ack* (buffer) para armazenar as mensagens já recebidas

```
def rdt_4_0_receive(self):
    self.byte_buffer = ''
    pack_ack = self.pack_ack
    ret_S = None
    ret_seq = None
    byte_S = self.network.udt_receive()
    self.byte_buffer += byte_S
    # keep extracting packets - if reordered, could get more than one
    while True:
        # check if we have received enough bytes
        if len(self.byte_buffer) < Packet.length_S_length:
            break # not enough bytes to read packet length
        # extract length of packet
        length = int(self.byte_buffer[:Packet.length_S_length])
        if len(self.byte_buffer) < length:
            break # not enough bytes to read the whole packet
```

Figura 20. Rdt Receiver

Ao receber uma mensagem, o código abaixo define qual deve ser a resposta a ser enviada para o Sender com o objetivo de manter o envio das mensagens fluida. Como é possível visualizar, um pacote corrompido receberá NACK, um caractere especial receberá a si mesmo como componente da mensagem e, aos demais casos, reenvia-se o pacote com a mensagem igual ao número de sequência do pacote recebido. Para os casos em que o pacote ainda não foi recebido, armazena-se em buffer. Caso já esteja em buffer, infere-se que aquele pacote não recebeu ACK no lado emissor.

```

1  if Packet.corrupt(self.byte_buffer):
2      if(Packet.corrupt(self.byte_buffer[0:length])):
3          # Pacote veio corrompido
4          debug_log("RECEIVER: Corrupt packet")
5          #Num seq Aleatorio, mensagem N
6          answer = Packet(0, "N")
7          self.network.udt_send(answer.get_byte_S())
8          break
9      debug_log("RECEIVER: Corrupt packet")
10     #Pacote veio corrompido
11     answer = Packet(Packet.from_byte_S(self.byte_buffer[0:length]).seq_num, "N")
12     self.network.udt_send(answer.get_byte_S())
13
14 else:
15     # create packet from buffer content
16     p = Packet.from_byte_S(self.byte_buffer[0:length])
17
18     if (p.msg_S == "\0"):
19         #Recebeu o caractere para parar de receber
20         debug_log("RECEIVER: END OF TRANSMISSION")
21         answer = Packet(p.seq_num, "\0")
22         self.network.udt_send(answer.get_byte_S())
23         #break
24
25     elif p.seq_num in pack_ack:
26         #ja recebeu esse pacote antes
27         debug_log(
28             'RECEIVER: Already received packet. ACK(n) again.')
29
30         answer = Packet(p.seq_num, f"{p.seq_num}")
31         self.network.udt_send(answer.get_byte_S())
32         #break
33
34     else:
35         debug_log(
36             'RECEIVER: Received new. Send ACK(n).')
37         # SEND ACK
38         answer = Packet(p.seq_num, f"{p.seq_num}")
39         self.network.udt_send(answer.get_byte_S())
40         pack_ack[p.seq_num] = p.seq_num
41         debug_log(f"{pack_ack}")

```

Figura 21. Controle da resposta

```

# Add contents to return string
ret_S = p.msg_S if (ret_S is None) else ret_S + p.msg_S
ret_seq = p.seq_num if (ret_seq is None) else ret_seq + p.seq_num
# remove the packet bytes from the buffer
self.byte_buffer = self.byte_buffer[length:]
# if this was the last packet, will return on the next iteration
# if(p.msg_S in pack_ack):
#     break

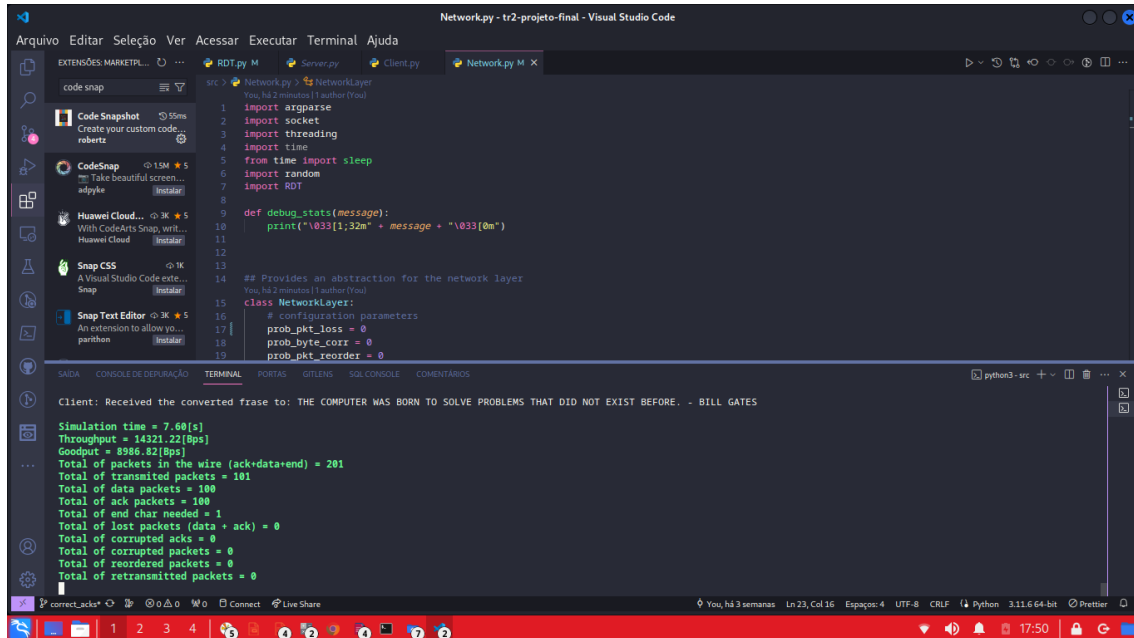
# remove the packet bytes from the buffer
self.byte_buffer = self.byte_buffer[length:]
# if this was the last packet, will return on the next iteration
if (ret_S):
    debug_log(f"RECEIVER: MSG_RECEIVED = {ret_S}")
return (ret_seq, ret_S)

```

Figura 22. Retorno do Método

3.4. Resultados:

Para efeitos de comparação, simularemos nossos resultados com o argumento de linha 19, que nos fornecerá 100 mensagens. Além disso, estabeleceremos um timeout de $0.4[s]$, o que agilizará a aplicação, mas pode a tornar um pouco mais propensa a erros.



```
src > Network.py > NetworkLayer
1 You, há 2 minutos | 1 author (You)
2 import argparse
3 import socket
4 import threading
5 import time
6 from time import sleep
7 import random
8 import RDT
9
10 def debug_stats(message):
11     print("\033[1;32m" + message + "\033[0m")
12
13
14 ## Provides an abstraction for the network layer
15 You, há 2 minutos | 1 author (You)
16
17 class NetworkLayer:
18     # configuration parameters
19     prob_pkt_loss = 0
20     prob_byte_corr = 0
21     prob_pkt_reorder = 0
22
23
24 Client: Received the converted frase to: THE COMPUTER WAS BORN TO SOLVE PROBLEMS THAT DID NOT EXIST BEFORE. - BILL GATES
25
26 Simulation time = 7.60[s]
27 Throughput = 14321.22[Bps]
28 Goodput = 8986.82[Bps]
29 Total of packets in the wire (ack+data+end) = 281
30 Total of transmitted packets = 101
31 Total of data packets = 100
32 Total of ack packets = 100
33 Total of end char needed = 1
34 Total of lost packets (data + ack) = 0
35 Total of corrupted acks = 0
36 Total of corrupted packets = 0
37 Total of reordered packets = 0
38 Total of retransmitted packets = 0
```

Figura 23. Simulação sem erro

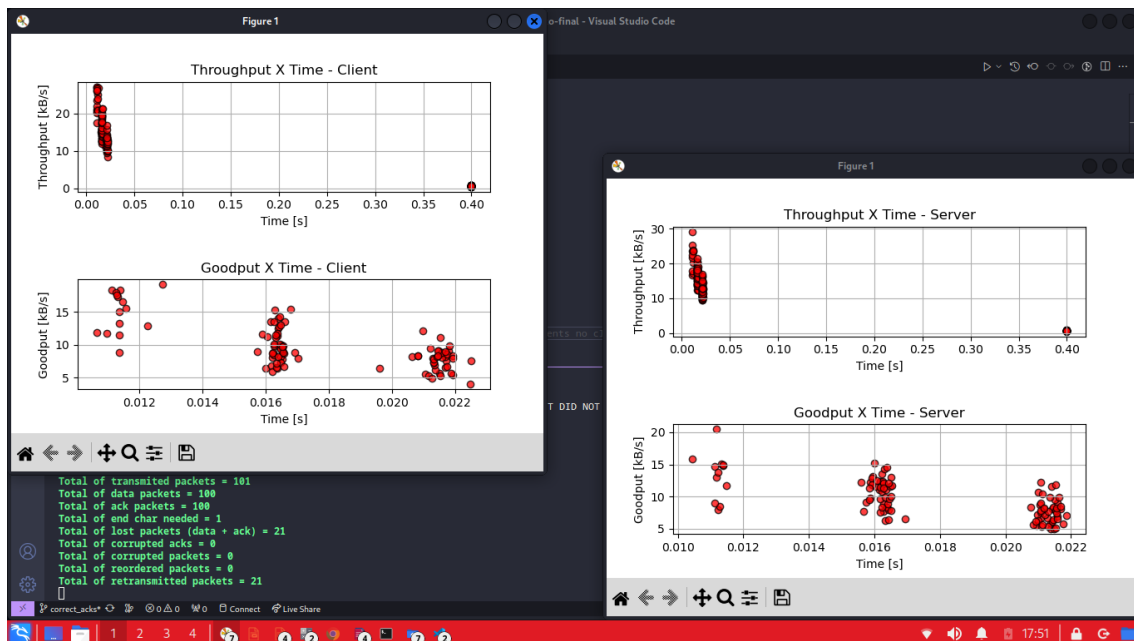


Figura 24. Simulação sem erro

Como se pode ver acima, não há perda ou pacotes corrompidos/reordenados, o que confere com o esperado.

```

src > Network.py > NetworkLayer
You, há 18 horas | 1 author (You)
1 import argparse
2 import socket
3 import threading
4 import time
5 from time import sleep
6 import random
7 import RDT
8
9 def debug_stats(message):
10     print("\033[1;32m" + message + "\033[0m")
11
12
13
14 ## Provides an abstraction for the network layer
15 You, há 18 horas | 1 author (You)
16 class NetworkLayer:
17     # configuration parameters
18     prob_pkt_loss = 0.1
19     prob_byte_corr = 0
20     prob_pkt_reorder = 0

```

Client: Received the converted frase to: THE COMPUTER WAS BORN TO SOLVE PROBLEMS THAT DID NOT EXIST BEFORE. - BILL GATES

```

Simulation time = 24.14[s]
Throughput = 3116.75[Bps]
Goodput = 1607.00[Bps]
Total of packets in the wire (ack+data+end) = 201
Total of transmitted packets = 101
Total of data packets = 100
Total of ack packets = 100
Total of end char needed = 1
Total of lost packets (data + ack) = 21
Total of corrupted acks = 0
Total of corrupted packets = 0
Total of reordered packets = 0
Total of retransmitted packets = 21

```

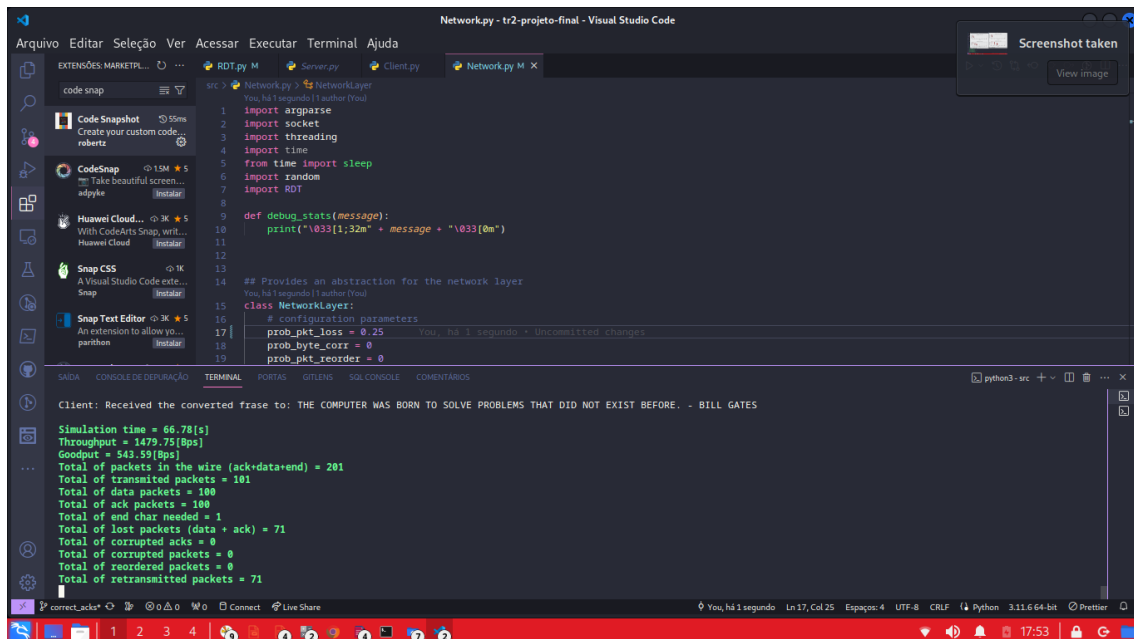
Figura 25. Simulação com 10% de Pkt Loss



Figura 26. Simulação com 10% de Pkt Loss

Como se pode verificar, houve perda de 21 dos 201 pacotes no total transmitidos no meio, o que significa um total de 10% dos pacotes, podendo variar entre, nessa

simulação em específico, ACKS ou packets, conferindo, novamente, com o esperado.



```
1 import argparse
2 import socket
3 import threading
4 import time
5 from time import sleep
6 import random
7 import rdt
8
9 def debug_stats(message):
10     print('\033[1;32m' + message + '\033[0m')
11
12
13
14 ## Provides an abstraction for the network layer
15 class NetworkLayer:
16     # configuration parameters
17     prob_pkt_loss = 0.25
18     prob_byte_corr = 0
19     prob_pkt_reorder = 0
```

Client: Received the converted frase to: THE COMPUTER WAS BORN TO SOLVE PROBLEMS THAT DID NOT EXIST BEFORE. - BILL GATES

Simulation time = 66.78[s]
Throughput = 1479.75[Bps]
Goodput = 543.59[Bps]
Total of packets in the wire (ack+data+end) = 201
Total of transmitted packets = 101
Total of data packets = 100
Total of ack packets = 100
Total of end char needed = 1
Total of lost packets (data + ack) = 71
Total of corrupted acks = 0
Total of corrupted packets = 0
Total of reordered packets = 0
Total of retransmitted packets = 71

Figura 27. Simulação com 25% de Pkt Loss

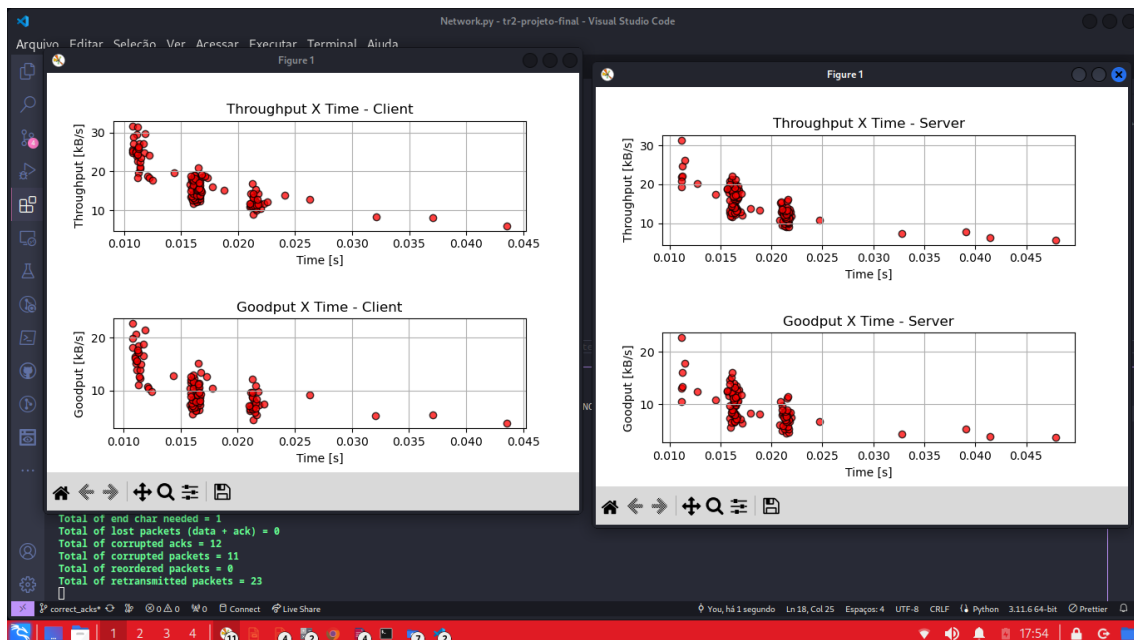
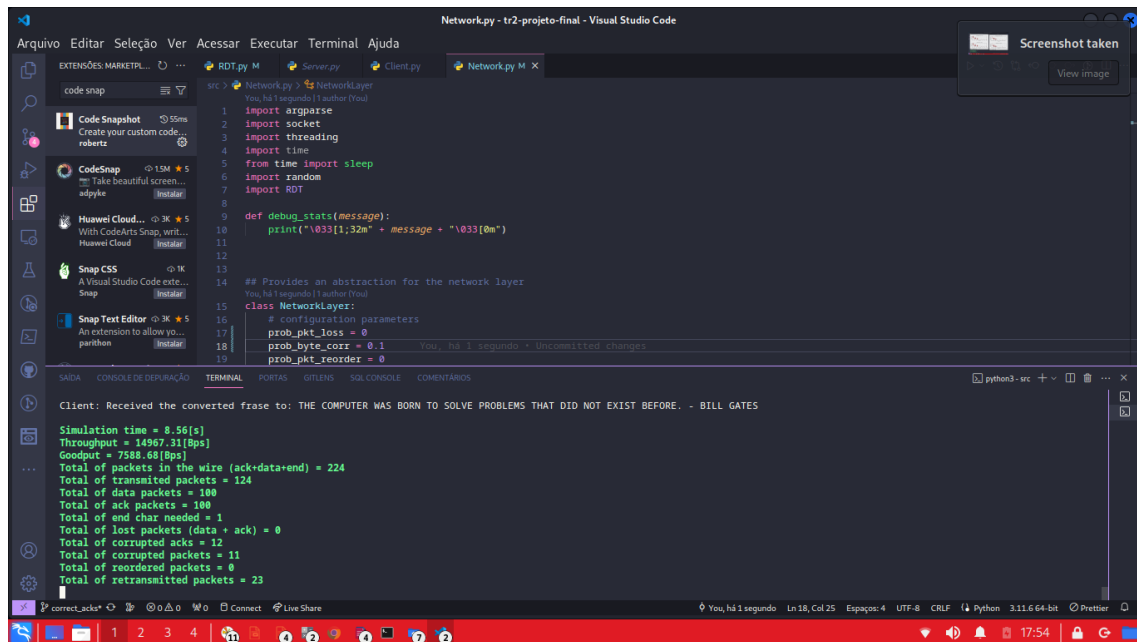


Figura 28. Simulação com 25% de Pkt Loss

Neste caso, houve a perda de 71 de 201 pacotes no total, o que significa cerca de 35% dos pacotes. É um valor um pouco acima do esperado, no entanto, não representa um quantitativo muito destoante do previsto, que seria de 51 pacotes perdidos. Porém,

na camada de rede, a probabilidade de perda pode ter afetado um pouco mais o sistema (algoritmo do código-base), o que pode ter prejudicado levemente os resultados.



```
src > Network.py > NetworkLayer
You, há 1 segundo | 1 author (You)
1 import argparse
2 import socket
3 import threading
4 import time
5 from time import sleep
6 import random
7 import RDT
8
9 def debug_stats(message):
10     print("\033[1;32m" + message + "\033[0m")
11
12
13
14 ## Provides an abstraction for the network layer
15 You, há 1 segundo | 1 author (You)
16 class NetworkLayer:
17     # configuration parameters
18     prob_pkt_loss = 0
19     prob_byte_corr = 0.1
20     prob_pkt_reorder = 0
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

Client: Received the converted frase to: THE COMPUTER WAS BORN TO SOLVE PROBLEMS THAT DID NOT EXIST BEFORE. - BILL GATES

Simulation time = 8.56[s]
Throughput = 14967.31[Bps]
Goodput = 7588.68[Bps]
Total of packets in the wire (ack+data+end) = 224
Total of transmitted packets = 124
Total of data packets = 100
Total of ack packets = 100
Total of end char needed = 1
Total of lost packets (data + ack) = 0
Total of corrupted acks = 12
Total of corrupted packets = 11
Total of reordered packets = 0
Total of retransmitted packets = 23

Figura 29. Simulação com 10% de Pkt Corrupt

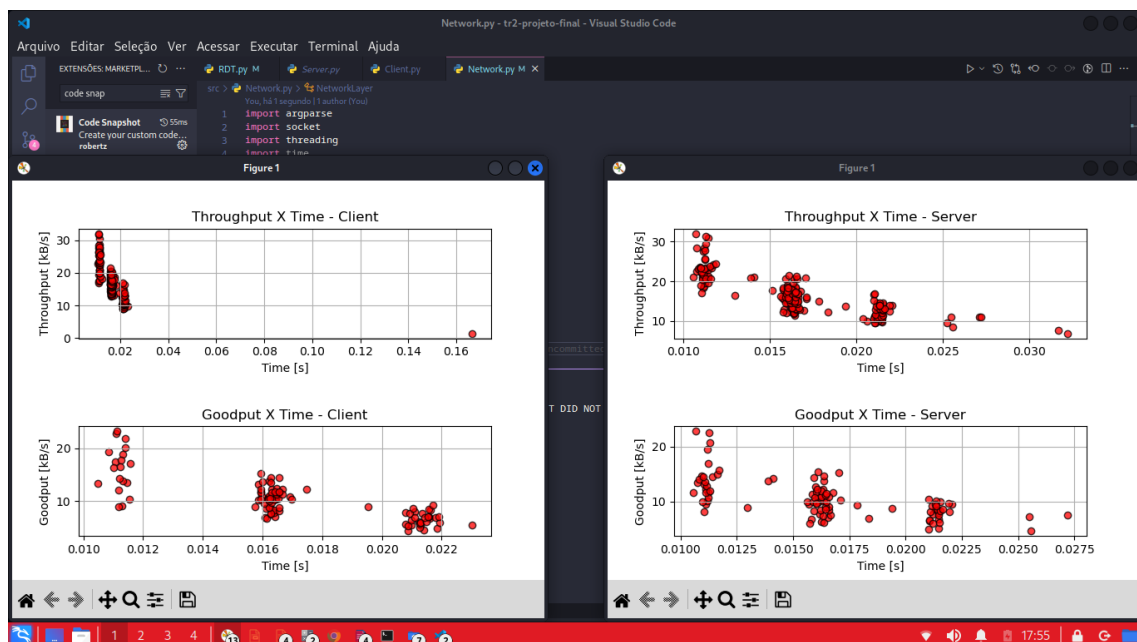


Figura 30. Simulação com 10% de Pkt Corrupt

No caso acima, houve, ao total, 23 pacotes corrompidos de 224, levando ao quantitativo de cerca de 10% de pacotes corrompidos, o que é conforme o esperado.

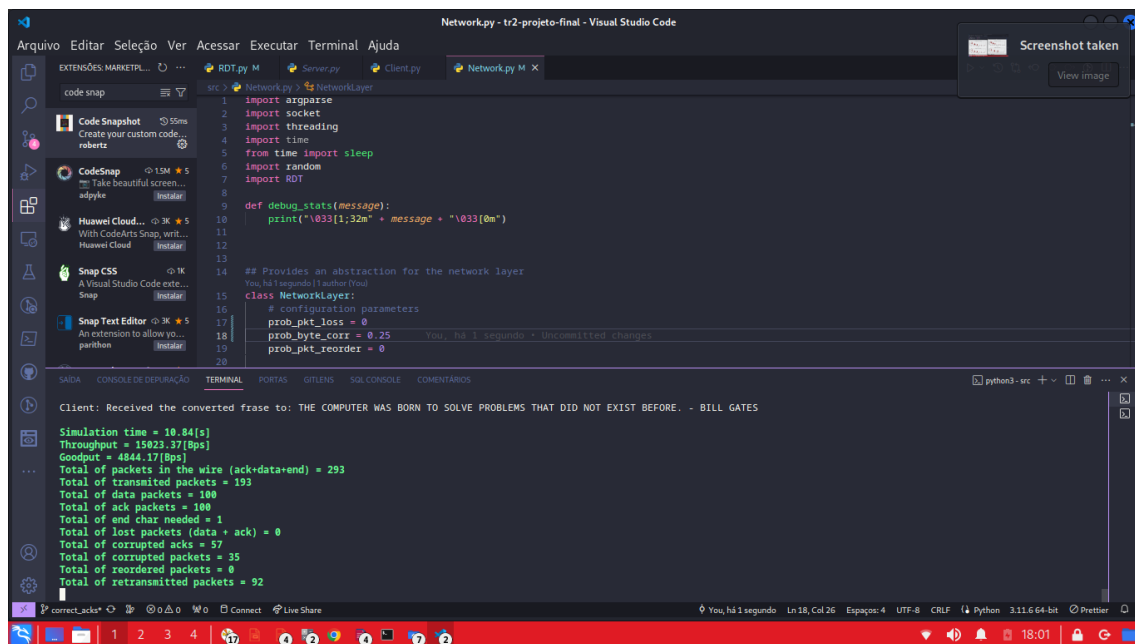


Figura 31. Simulação com 25% de Pkt Corrupt

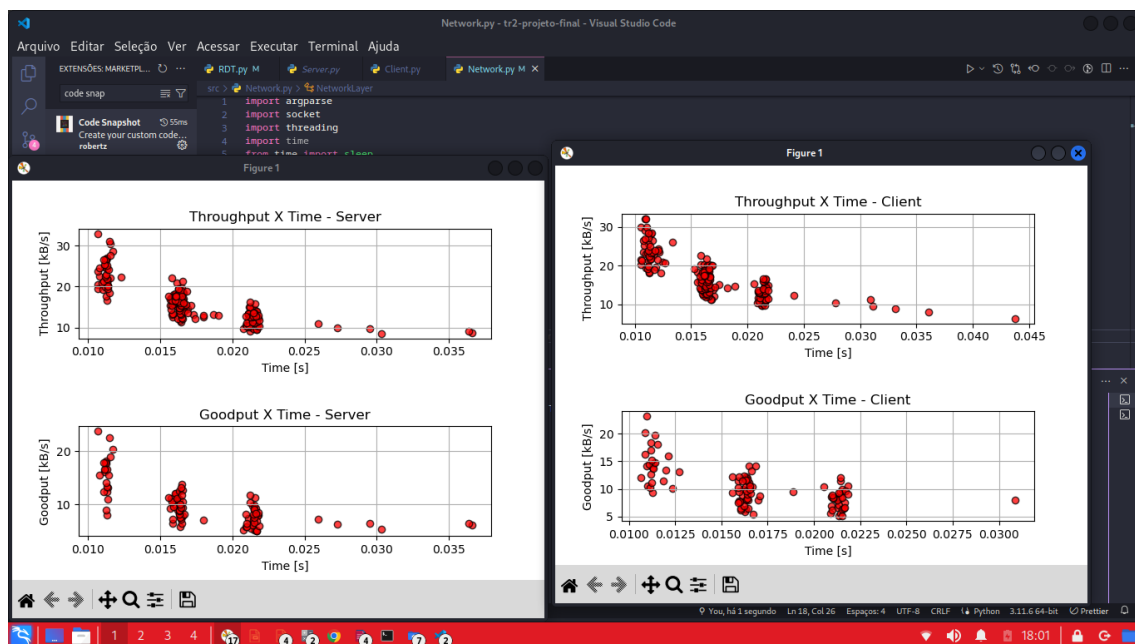


Figura 32. Simulação com 25% de Pkt Corrupt

No caso de 25%, houve cerca de 92 pacotes corrompidos de um total de 293, representando cerca de 31% da amostragem total. Novamente, o valor foi um pouco acima do esperado, mas a aplicação pode ter sofrido efeitos probabilísticos na camada de rede que afetaram levemente os resultados finais.

Extra: Estatísticas no Servidor

Não nos foi requisitado, mas iremos promover um exemplo da janela de acordo com seu espaçamento de 5 pacotes.

```

15 class NetworkLayer:
16     # configuration parameters
17     prob_pkt_loss = 0.1
18     prob_byte_corr = 0
19     prob_pkt_reorder = 0

```

SAÍDA CONSOLE DE DEBURAÇÃO TERMINAL PORTAS UTILIZADAS SQL CONSOLE COMENTÁRIOS python3 -src + -

NEW PACKET
 SENDER: ACK received
 SENDER: TRANSMITTING PACKET -> THE COMPUTER WAS BORN TO SOLVE PROBLEMS THAT DID NOT EXIST BEFORE. - BILL GATES
 PACK_ACK == (0: '0', 1: '1', 2: '2', 3: '3', 4: '4', 5: '5', 6: '6', 7: '7', 8: '8', 9: '9', 10: '10', 11: '11', 12: '12', 13: '13', 14: '14', 15: '15', 16: '16', 17: '17', 18: '18', 19: '19', 20: '20', 21: '21', 22: '22', 23: '23', 24: '24', 25: '25', 26: '26', 27: '27', 28: '28', 29: '29', 30: '30', 31: '31', 32: '32', 33: '33', 34: '34', 35: '35', 36: '36', 37: '37', 38: '38', 39: '39', 40: '40', 41: '41', 42: '42', 43: '43', 44: '44', 45: '45', 46: '46', 47: '47', 48: '48', 49: '49', 50: '50', 51: '51', 52: '52', 53: '53', 54: '54', 55: '55', 56: '56', 57: '57', 58: '58', 59: '59', 60: '60', 61: '61', 62: '62', 63: '63', 64: '64', 65: '65', 66: '66', 67: '67', 68: '68', 69: '69', 70: '70', 71: '71', 72: '72', 73: '73', 74: '74', 75: '75', 76: '76', 77: '77', 78: '78', 79: '79', 80: '80', 81: '81', 82: '82', 83: '83', 84: '84', 85: '85', 86: '86', 87: '87', 88: '88', 89: '89', 90: '90', 91: '91', 92: '92', 93: '93', 94: '94', 95: '95', 96: '96', 97: '97', 98: '98')

Figura 35. Sliding Window

Conforme a imagem acima, vê-se muitos tracejados vermelhos que indicam uma quebra da progressão aritmética do número de sequência, mas isto pode ser explicado pelo uso do Selective Repeat.

A primeira quebra ocorre na sequência de recebimento de ACKS 4,5,8. Isso quer dizer que a nova janela será 6,7,8,9,10, já que a base da janela (6) não foi recebida ainda, tampouco 7. Como o 8 já foi recebido, ele não será reenviado. Portanto, pode-se até enviar o 7, o 9 e o 10, mas a janela não progride sem o envio do 6, como se vê na simulação; o 9 é enviado antes mesmo do 6, que, após ser enviado, desloca a base da janela para 7, que ficará 7,8,9,10,11, e assim sucessivamente para os outros casos, comprovando o devido funcionamento da janela deslizante conforme o protocolo selective repeat.

Reorder Messages

Promovemos simulações com probabilidade de packet reorder de 10% e 25%.

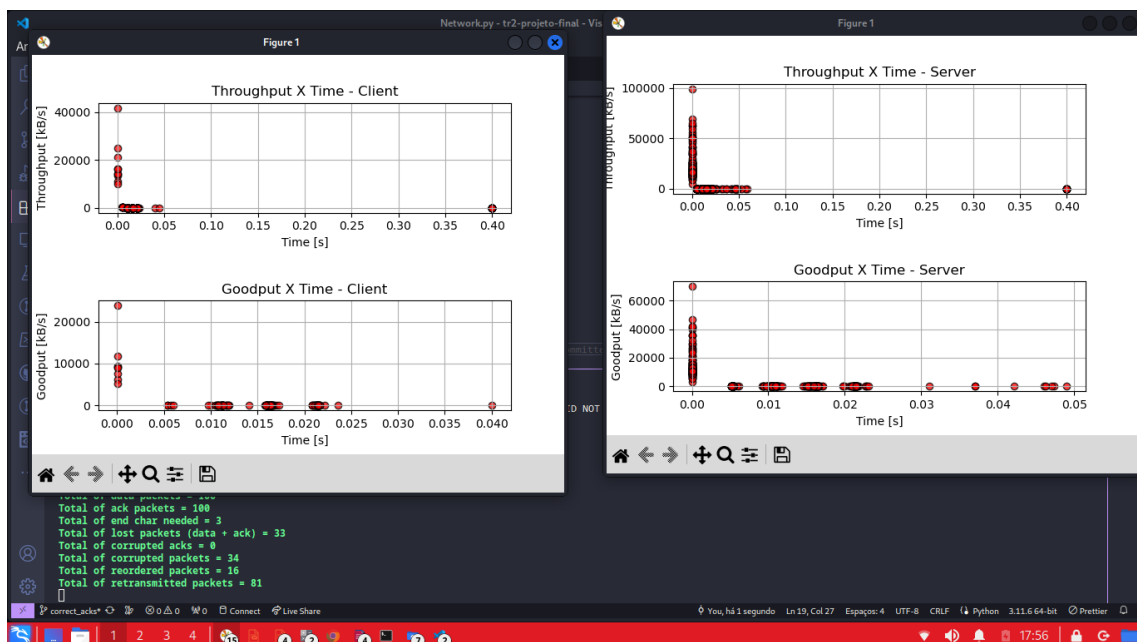


Figura 36. Simulação com 10% de Pkt Reorder

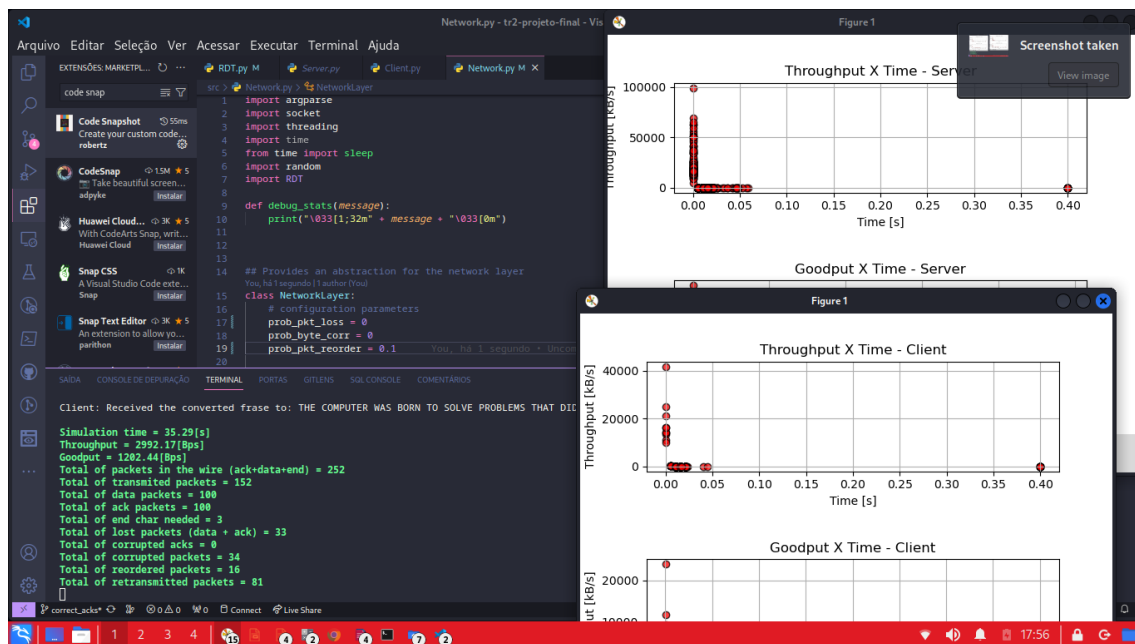


Figura 37. Simulação com 10% de Pkt Reorder

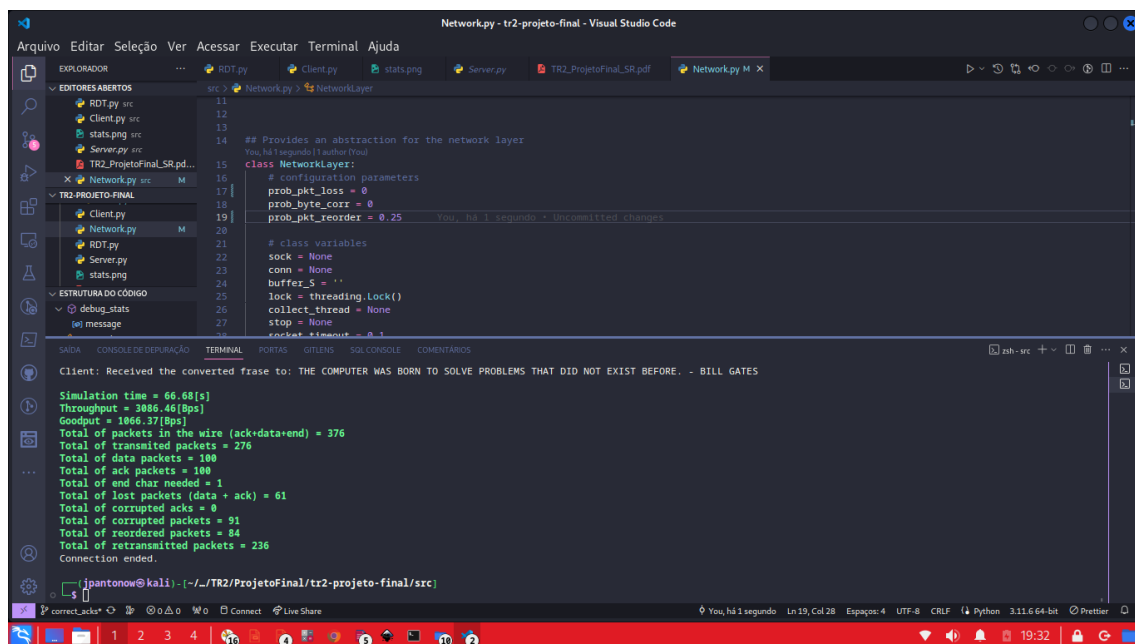


Figura 38. Simulação com 25% de Pkt Reorder

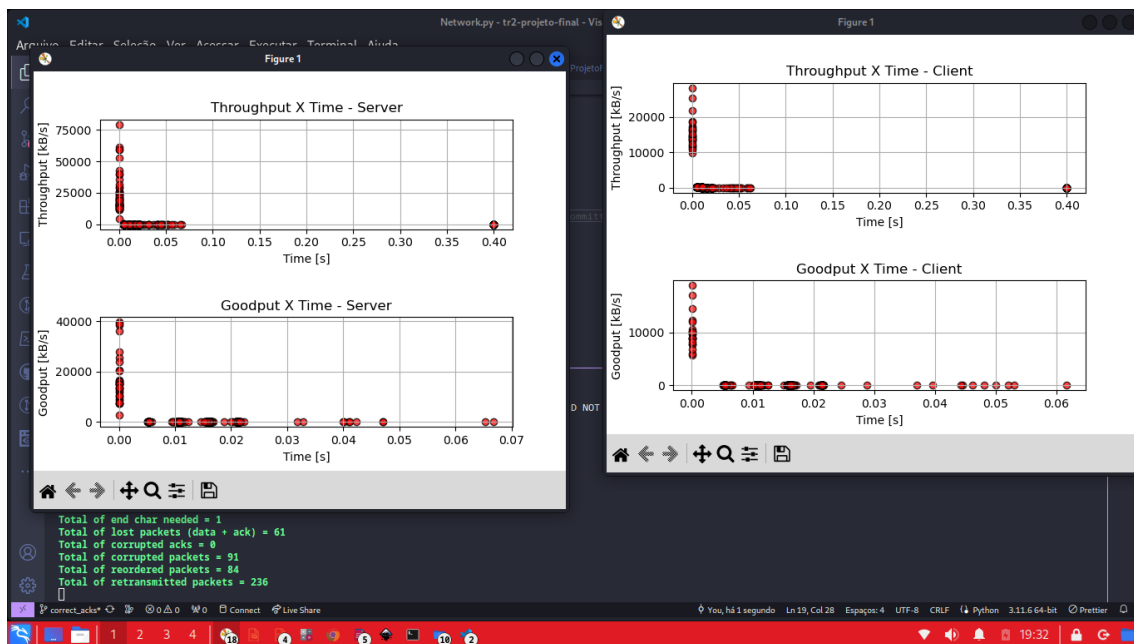


Figura 39. Simulação com 25% de Pkt Reorder

O valor de 16 amostras possivelmente reordenadas de um total de 252 representa cerca de 6%, ou seja, um pouco abaixo do esperado; a segunda amostra possui 84 pacotes reordenados de um total de 376, conferindo 22% de reordenação (também um pouco abaixo). Contudo, deve-se salientar, novamente, o efeito prejudicial dos algoritmos do código-base no levantamento de estatísticas.

```
# reorder packets - either hold a packet back, or if one held back then send both
if random.random() < self.prob_pkt_reorder or self.reorder_msg_S:
    if self.reorder_msg_S is None:
        self.reorder_msg_S = msg_S
        return None
    else:
        msg_S += self.reorder_msg_S
        self.reorder_msg_S = None
# keep calling send until all the bytes are transferred
```

Figura 40. Algoritmo de Reordenação - Network

Como visto acima, a concatenação de strings leva, em muitos casos, à interpretação de que um pacote ou ack está corrompido, o que prejudica a interpretação deste por parte do RDT sender ou RDT receiver, causando alguns desvios na amostra, que interpretará, em alguns casos, como pacote corrompido. Além disso, a reordenação pode levar a uma resposta nula, o que será interpretado como Lost Packet. Dado esses fatos, considera-se, apesar disso, que o valor encontrado para os pacotes reordenados está dentro do esperado.

4. Conclusão

Os resultados do projeto ocorreram como o esperado para o selective repeat, visto que as métricas condizem com os resultados esperados causados pelas interferências no envio das mensagens. O código apresenta um bom nível de eficiência para o envio e retorno das mensagens no quesito tempo, assim como, não apresenta timeouts desnecessários que poderiam causar overheads na transmissão. Seguindo essa ideia, é possível afirmar que os métodos utilizados para a formulação do RDT 4.0 garantiram uma boa segurança e efetividade da transmissão.

Referências

- [Sah 2023] (2023). Sliding window simulator. <https://nikhilsahu.me/Sliding-window-simulator/>.
- [Kurose 2016] Kurose (2016). *Computer Networking: A Top-Down Approach*.