

Projeto 2 - Implementar o Protocolo TCP

Grupo:

Guilherme Nishi Kanashiro - 628298

Leonardo Utida Alcântara - 628182

Rodolfo Krambeck Asbahr - 628042

Tiago Bachiega de Almeida - 628247

Como executar o programa:

A implementação do protocolo TCP foi feita utilizando python 3. Para rodar o código em um terminal é preciso antes executar o seguinte comando, para evitar que o linux feche as conexões TCP abertas pelo programa:

```
$ sudo iptables -I OUTPUT -p tcp --tcp-flags RST RST -j DROP
```

Em seguida para rodar o programa basta executar o seguinte comando:

```
$ sudo python3 projeto2.py
```

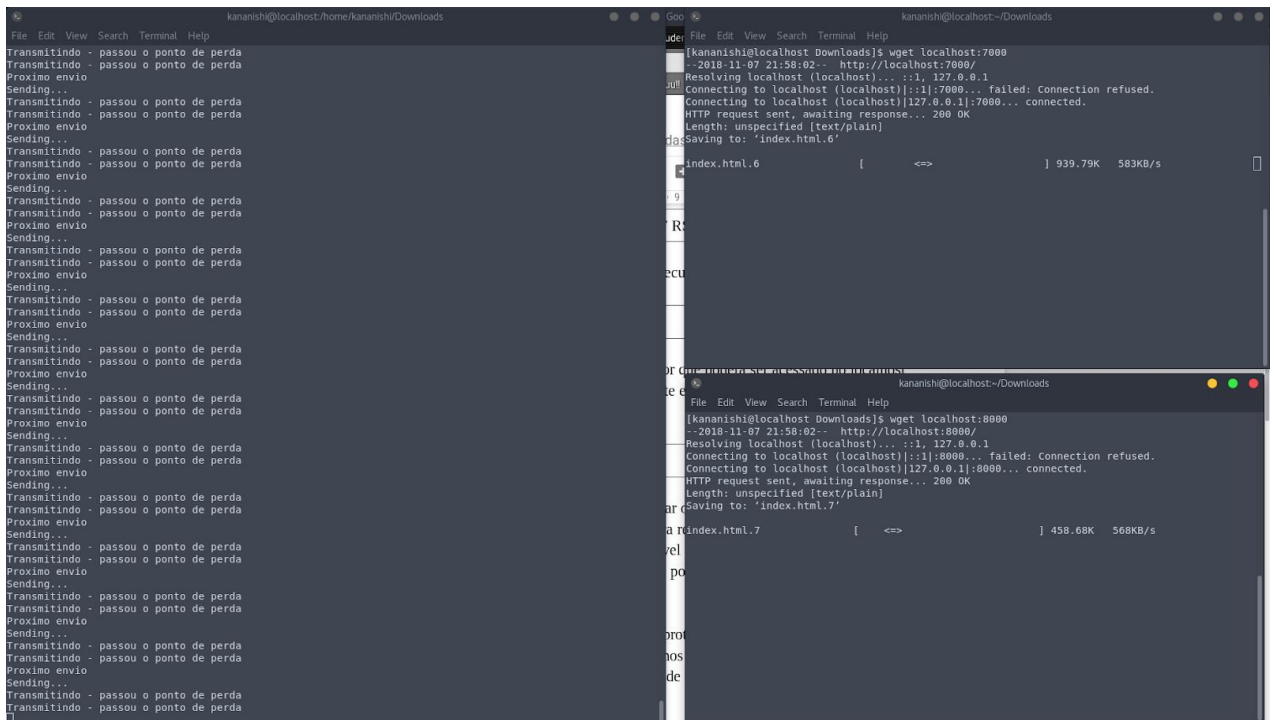
Feito isso, o programa irá iniciar um servidor que poderá ser acessado no localhost usando a porta 7000 ou 8000. Para iniciar um cliente e estabelecer uma conexão com o servidor basta executar o comando abaixo:

```
$ wget localhost:7000
```

Para testar o controle de fluxo é possível usar o comando Ctrl+z no terminal para parar o processo do wget e depois usar o comando fg para retomar o processo.

Para testar o controle de congestão, é possível usar executar em dois terminais o comando wget usando a porta 7000 no primeiro e a porta 8000 na segunda. Na Figura 1, abaixo, pode-se verificar a divisão de banda.

Figura 1 - Código rodando, mostrando o controle de fluxo



```
kananishi@localhost:~/Downloads$ wget localhost:7000
--2018-11-07 21:58:02-- http://localhost:7000/
Resolving localhost (localhost)... ::1, 127.0.0.1
Connecting to localhost (localhost)::1:7000... failed: Connection refused.
Connecting to localhost (localhost)|127.0.0.1|:7000... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/plain]
Saving to: 'index.html.6'

index.html.6           [ <-> ] 939.79K  583KB/s

kananishi@localhost:~/Downloads$ wget localhost:8000
--2018-11-07 21:58:02-- http://localhost:8000/
Resolving localhost (localhost)... ::1, 127.0.0.1
Connecting to localhost (localhost)::1:8000... failed: Connection refused.
Connecting to localhost (localhost)|127.0.0.1|:8000... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/plain]
Saving to: 'index.html.7'

index.html.7           [ <-> ] 458.68K  568KB/s
```

Como funciona o programa:

O código funciona baseado na classe **Conexão** e na implementação da **Máquina de Estados**. A classe **Conexão** possui uma série de componentes, como por exemplo o *id_conexao*, *seq_no*, *ack_no*, *RTO*, *send_queue*, entre outros. A Figura 2 mostra como a classe está implementada no programa e uma breve descrição do que é cada parâmetro.

Figura 2 - Classe Conexao e seus parâmetros

```
class Conexao:
    def __init__(self, id_conexao, seq_no, ack_no):
        self.id_conexao = id_conexao #ID da conexao
        self.seq_no = seq_no #num de sequencia
        self.ack_no = ack_no #ACK
        self.send_base = seq_no # base
        self.first_ack = True # significa que o handshake ainda não acabou
        self.timer = None # objeto timer da conexao
        self.start_t = 0 # tempo de inicio da conexao
        self.end_t = 0 # tempo de fim da conexao
        self.RTO = 3.0 # timeout
        self.last_RTT = 0.0 # ultimo RTT
        self.curr_RTT = 0.0 # RTT atual
        self.rwnd = self.cwnd = 2*MSS #receiver window e congestion window
        self.rx_win = 1024 # janela de transmissao
        self.ssthresh = 10*MSS # janela limite
        self.state = "Slow Start" # estado par a maquina de estados
        self.last_ack = self.send_base #ultimo ack aceito
        self.send_queue = b"HTTP/1.0 200 OK\r\nContent-Type: text/plain\r\n\r\n" + 100000 * b"hello pombo\n"
        self.no_ack_queue = b""
        self.sent = [] #enviados
        self.flag_fin = False # flag fin
        self.flag_handshake = True # handshake
        self.flag_close_conection = False # fechar conexao
        self.store_time = {} # armazena os times
conexoes = {}
```

Os objetos **Conexão** são utilizados em praticamente todas as funções do código. Foram criadas algumas funções para facilitar a manipulação dos envios, recebimentos e das criações de sinais, como *cria_synack()*, *cria_ack()*, *cria_fin()*, *enviaSrc()* e *enviaDst()*. A função *transmit_as_allowed()* é responsável por realizar a transmissão de dados e sinais propriamente dita. Ela também possui um trecho responsável por **testar o envio incompleto** de pacotes, trecho este mostrado na Figura 3.

Figura 3 - Simula a perda de pacote

```
if not TESTAR_PERDA_ENVIO or random.random() < 0.95:  
    print("Transmitindo - passou o ponto de perda")  
    enviaDst(fd, conexao, segment)
```

A função *ack_recv()* foi criada para o tratamento do *ack_no*. Esta função trata o **handshake inicial**, o **handshake final** de acordo com a flag **FIN** e os demais **ACKs recebidos** durante a transmissão. Por sua vez, a função *RTO* calcula o **timeout adaptativo para retransmissão do pacote** de acordo com a RFC 2988.

A função *retransmission()* trata a **retransmissão de pacotes perdidos** de acordo com o estado atual da **Máquina de Estados** implementada. Vale a pena observar que o tratamento dos parâmetros *conexao.ssthresh*, *conexao.cwnd* e *conexao.dup_ack_cnt* são idênticos nos estados **Slow Start** e **Congestion Avoidance**. No estado **Fast Recovery** há uma pequena diferença no valor dado a *conexao.cwnd*.

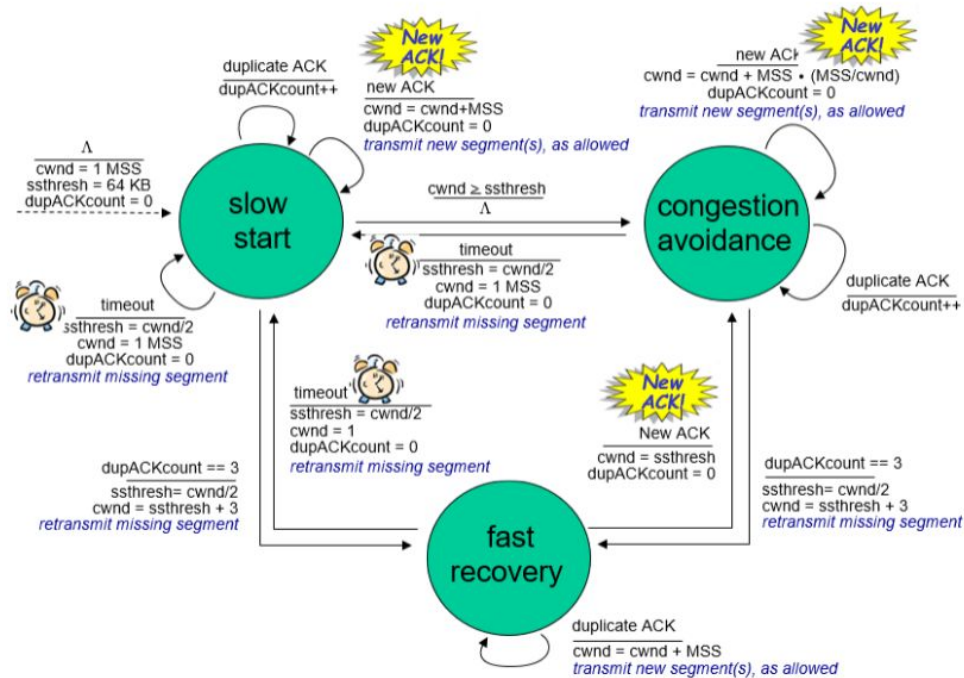
Por fim, temos a função *raw_recv()*, que efetivamente implementa a **Máquina de Estados** da Figura 4 e realiza a maior parte do processamento do programa. O programa inicia-se no estado **Slow Start**. Nele é feito o **handshaking**, o recebimento de um **novo ACK**, o recebimento de **ACKs duplicados** e o tratamento do **timeout**. Caso *conexao.cwnd* seja maior ou igual a *conexao.ssthresh*, o estado muda para **Congestion Avoidance**. Caso o número de **ACKs duplicados** seja 3, o estado muda para **Fast Recovery**.

No estado **Congestion Avoidance** é feito o tratamento do **congestionamento do código**, operando sobre o parâmetro *conexao.cwnd*. Neste estado ocorrem os tratamentos para um **novo ACK recebido**, para **ACKs duplicados** e **timeout**. Caso ocorra **timeout**, o estado volta para **Slow Start**. Caso ocorram 3 **ACKs duplicados**, o estado vai para **Fast Recovery**.

Em **Fast Recovery** ocorre o tratamento para que o programa não fique estagnado esperando que sejam resolvidos **ACKs duplicados** que não conseguem se resolver. Para isso, caso receba um **ACK duplicado**, ele apenas vai para o próximo dado a ser transmitido e realiza a transmissão, ignorando a transmissão dos dados não confirmados. Em caso de **timeout**, o estado vai para **Slow Start**. Em caso de **novo ACK**, o estado vai para **Congestion Avoidance**.

Figura 4 - Máquina de Estados

Summary: TCP Congestion Control



O que funciona e o que não funciona:

- O programa estabelece conexão com número de sequência aleatório.
- O programa transmite e recebe corretamente os segmentos.
- O programa consegue identificar e tratar parcialmente as retransmissões. O programa somente consegue retransmitir corretamente o último pacote perdido e detectado a tempo. Não foi implementada a identificação de qual pacote em específico deve ser retransmitido, ele sempre retransmite o último.
- Foi implementado o cálculo do timeout adaptativo para retransmissão, embora não tenhamos certeza de que ele funcione quando há mais de uma conexão ocorrendo.
- Foi implementada toda a Máquina de Estados para os timeouts e ACKs duplos.
- Foi implementado o controle de congestionamento de acordo com a Máquina de Estados no estado Congestion Avoidance.
- Foi implementado o controle de fluxo embora não tenhamos certeza se foi feito do jeito correto e completamente.
- A conexão está sendo fechada de forma limpa com a flag FIN.