Protocolo UDP Modificado com Controle de Congestionamento e Criptografia

Documentação Técnica do Projeto
Redes de Computadores
Implementação de Protocolo de Transporte Confiável

30 de janeiro de 2026

Este documento descreve a implementação de um protocolo de transporte confiável sobre UDP que incorpora mecanismos de confiabilidade, controle de fluxo e congestionamento. O protocolo utiliza números de sequência para entrega ordenada, ACKs acumulativos, janela de congestionamento (cwnd) com fases Slow Start e Congestion Avoidance, e criptografia simétrica para proteção dos dados. A avaliação experimental demonstra a eficácia dos mecanismos implementados.

1 Introdução

O UDP (User Datagram Protocol) é um protocolo de transporte não confiável que não garante entrega, ordenação ou controle de congestionamento. Este projeto implementa um protocolo customizado sobre UDP que adiciona características de confiabilidade similares ao TCP, incluindo:

- Estabelecimento e encerramento de conexão (handshake)
- Entrega ordenada de pacotes com números de sequência
- Confirmações acumulativas (ACKs)
- Controle de fluxo baseado em janela de recepção
- Controle de congestionamento com cwnd e ssthresh
- Criptografia de dados (Caesar Cipher)

Os arquivos principais da implementação são `server_final.py` (remetente) e `client_final.py` (destinatário).

2 Requisito 1: Entrega Ordenada com Números de Sequência

2.1 Localização no Código

A entrega ordenada é implementada no arquivo `client_final.py`, função `receive_and_ack()`, linhas 121-228.

2.2 Implementação

Cada pacote transmitido possui um campo `seq` (número de sequência) que indica a posição do byte inicial no fluxo de dados:

```
pct_zero = {
    'seq'       : 0,
    'ack'       : 0,
    'rwnd'      : 1024,
    'SYN'       : False,
    'FIN'       : False,
    'payload'   : ""
}
```

O servidor incrementa o número de sequência pelo tamanho do payload (em bytes):

Listing 1: server_final.py - linhas 123-135

```
payload = f"Mensagem numero {next_msg}"
payload_size = len(caesar_cipher(payload).encode('utf-8'))

pct = pct_zero.copy()
pct['seq'], pct['payload'] = current_seq, payload
...
current_seq += payload_size
```

O cliente mantém uma variável `expected_seq` que rastreia o próximo byte esperado e um buffer `out_of_order_buffer` para pacotes fora de ordem:

Listing 2: client_final.py - linhas 125-135

```
expected_seq = initial_ack
out_of_order_buffer = {}  # {seq: (payload, payload_size)}

if seq == expected_seq:
    # PACOTE EM ORDEM
    received_count += 1
    expected_seq += payload_size
    pcts_since_ack += 1
    process_buffered_packets()
```

2.3 Buffer de Reordenação

Pacotes que chegam fora de ordem são armazenados no buffer até que os pacotes faltantes sejam recebidos:

Listing 3: client_final.py - linhas 186-200

```
elif seq > expected_seq:
    # PACOTE FORA DE ORDEM
    if seq not in out_of_order_buffer:
        out_of_order_buffer[seq] = (payload, payload_size)
        buffered_count += 1
        print(f"[!] Fora de ordem (bufferizado)")
```

A função `process_buffered_packets()` processa os pacotes bufferizados quando a lacuna é preenchida:

Listing 4: client_final.py - linhas 158-168

```
def process_buffered_packets():
    while expected_seq in out_of_order_buffer:
        payload, payload_size = out_of_order_buffer[expected_seq]
        del out_of_order_buffer[expected_seq]

        received_count += 1
        print(f"[BUFFER] Processando seq={expected_seq}")

        expected_seq += payload_size
        pcts_since_ack += 1
```

3 Requisito 2: Confirmação Acumulativa (ACK)

3.1 Localização no Código

ACKs acumulativos estão implementados em `client_final.py` (linhas 143-153) e `server_final.py` (linhas 139-160).

3.2 Implementação no Cliente

O cliente envia ACKs acumulativos indicando o próximo byte esperado:

Listing 5: client_final.py - linhas 143-153

```
def send_ack(reason=""):
    ack_pct = pct_zero.copy()
    ack_pct["ack"] = expected_seq  # ACK acumulativo
    ack_pct["seq"] = last_ack

    my_encode_and_send(connection, ack_pct, address)
    ack_sent_count += 1

    info = f"ACK (ack={expected_seq})"
    print(f"|------- {info} ------>| ({reason})")
```

3.3 Processamento de ACKs no Servidor

O servidor processa ACKs acumulativos, removendo todos os pacotes confirmados da janela:

Listing 6: server_final.py - linhas 139-151

```
if received_ack > base_seq:
    # ACK novo
    print(f"ACK (ack={received_ack})")

    # Remove todos os pacotes confirmados
    confirmed = [s for s in in_flight if s < received_ack]
    for seq in confirmed:
        del in_flight[seq]

    # Atualiza janela de congestionamento
    cwnd, ssthresh, in_fast_recovery, duplicate_acks = \
        handle_new_ack(cwnd, ssthresh, len(confirmed),
                       in_fast_recovery)

    base_seq = received_ack
```

3.4 ACKs Duplicados

ACKs duplicados são detectados e usados para Fast Retransmit:

Listing 7: server_final.py - linhas 153-172

```
elif received_ack == base_seq and in_flight:
    # ACK duplicado
    print(f"ACK (ack={received_ack}) (Duplicado)")

    cwnd, ssthresh, duplicate_acks, in_fast_recovery,
        should_retransmit = handle_duplicate_ack(
            cwnd, ssthresh, duplicate_acks, in_fast_recovery)

    if should_retransmit and in_flight:
        retrans_seq = min(in_flight.keys())
        msg_num, payload, _ = in_flight[retrans_seq]

        # Retransmite o pacote mais antigo
        pct = pct_zero.copy()
        pct['seq'], pct['payload'] = retrans_seq, payload
        my_encode_and_send(sock, pct, addr)

        retransmissions += 1
        print(f"RETRANS (seq={retrans_seq})")
```

4 Requisito 3: Controle de Fluxo

4.1 Localização no Código

O controle de fluxo está implementado em `server_final.py`, função `get_window_size()`, linhas 60-61.

4.2 Janela de Recepção (rwnd)

O cliente anuncia sua janela de recepção disponível em cada ACK:

Listing 8: client_final.py - estrutura do pacote

```
pct_zero = {
    'rwnd'      : 1024,  # Janela de recepcao
    ...
}
```

4.3 Tamanho Efetivo da Janela

O servidor calcula o tamanho efetivo da janela como o mínimo entre cwnd (controle de congestionamento) e rwnd (controle de fluxo):

Listing 9: server_final.py - linhas 60-61

```
def get_window_size(cwnd, rwnd):
    return min(cwnd, rwnd)
```

O servidor respeita este limite ao enviar pacotes:

Listing 10: server_final.py - linhas 117-122

```
window_size = get_window_size(cwnd, last_rwnd)

# Envio inicial - respeita o tamanho da janela
while len(in_flight) < window_size and next_msg < total_msgs:
    payload = f"Mensagem numero {next_msg}"
    # ... envia pacote
```

O valor de `rwnd` é extraído dos ACKs recebidos:

Listing 11: server_final.py - linha 137

```
last_rwnd = ack_msg.get('rwnd', 1024)
```

5 Requisito 4: Controle de Congestionamento

5.1 Localização no Código

O controle de congestionamento está implementado em `server_final.py`, linhas 60-92.

5.2 Variáveis de Controle

O protocolo utiliza as seguintes variáveis, inspiradas no TCP:

Listing 12: server_final.py - linhas 14-18

```
initial_cwnd = 1.0        # Janela de congestionamento inicial
initial_ssthresh = 8      # Limiar de slow start inicial
max_cwnd = 32             # Janela maxima
timeout = 2.0             # Timeout para retransmissao
duplicate_ack_threshold = 3  # 3 ACKs duplicados
```

5.3 Slow Start

Durante a fase Slow Start (cwnd < ssthresh), a janela cresce exponencialmente:

Listing 13: server_final.py - linhas 63-68

```
def handle_new_ack(cwnd, ssthresh, num_packets,
                   in_fast_recovery):
    if in_fast_recovery:
        return ssthresh, ssthresh, False, 0
    elif cwnd < ssthresh:  # SLOW START
        return min(cwnd + num_packets, max_cwnd), ssthresh,
               False, 0
```

A janela aumenta em 1 MSS por cada ACK recebido, resultando em crescimento exponencial (duplicação a cada RTT).

5.4 Congestion Avoidance

Quando cwnd $\geq$ ssthresh, entra em Congestion Avoidance com crescimento linear:

Listing 14: server_final.py - linhas 68-70

```
    else:  # CONGESTION AVOIDANCE
        return min(cwnd + (num_packets / cwnd), max_cwnd),
               ssthresh, False, 0
```

O crescimento é de aproximadamente 1 MSS por RTT.

5.5 Fast Retransmit e Fast Recovery

Após 3 ACKs duplicados, o protocolo executa Fast Retransmit:

Listing 15: server_final.py - linhas 72-88

```
def handle_duplicate_ack(cwnd, ssthresh, duplicate_acks,
                         in_fast_recovery):
    duplicate_acks += 1
    should_retransmit = False

    if duplicate_acks == duplicate_ack_threshold and \
       not in_fast_recovery:
        # FAST RETRANSMIT
        ssthresh = max(int(cwnd / 2), 2)
        cwnd = ssthresh
        in_fast_recovery = True
        should_retransmit = True

        print(f"FAST RETRANSMIT - CWND: {cwnd:.1f}, "
              f"SSThresh: {ssthresh}")

    elif in_fast_recovery:  # FAST RECOVERY
        cwnd = min(cwnd + 1, max_cwnd)

    return cwnd, ssthresh, duplicate_acks, in_fast_recovery,
           should_retransmit
```

5.6 Tratamento de Timeout

Timeout resulta em redução drástica da janela:

Listing 16: server_final.py - linhas 90-97

```
def handle_timeout(cwnd, ssthresh):
    print(f"TIMEOUT - CWND: {cwnd:.1f} -> {initial_cwnd}, "
          f"SSThresh: {ssthresh} -> {max(int(cwnd/2), 2)}")

    return initial_cwnd, max(int(cwnd / 2), 2), False, 0
```

Após timeout:

- cwnd ← 1 MSS (volta ao início)
- Sai de Fast Recovery

6 Requisito 5: Criptografia

6.1 Localização no Código

A criptografia está implementada em ambos os arquivos:

- `client_final.py`: linhas 24-33
- `server_final.py`: linhas 22-31

6.2 Algoritmo: Caesar Cipher

Foi implementada a Cifra de César com deslocamento de 3 posições:

Listing 17: Função de criptografia

```
def caesar_cipher(text, shift=3):
    encrypted_text = ""
    for char in text:
        if 'a' <= char <= 'z':
            encrypted_text += chr((ord(char) - ord('a') + shift)
                                  % 26 + ord('a'))
        elif 'A' <= char <= 'Z':
            encrypted_text += chr((ord(char) - ord('A') + shift)
                                  % 26 + ord('A'))
        else:
            encrypted_text += char
    return encrypted_text

def caesar_decipher(text, shift=3):
    return caesar_cipher(text, -shift)
```

6.3 Aplicação da Criptografia

A criptografia é aplicada automaticamente em todas as transmissões através das funções wrapper:

Listing 18: client_final.py - linhas 40-52

```
def my_encode_and_send(socket, msg, adress_port):
    msg_copy = msg.copy()
    if 'payload' in msg_copy and msg_copy['payload']:
        msg_copy['payload'] = caesar_cipher(msg_copy['payload'])
    socket.sendto(json.dumps(msg_copy).encode('utf-8'),
                  adress_port)

def my_receive_and_decode(socket, buffer_size):
    pct, address = socket.recvfrom(buffer_size)
    msg = json.loads(pct.decode('utf-8'))
    if 'payload' in msg and msg['payload']:
        msg['payload'] = caesar_decipher(msg['payload'])
    return msg, address
```

6.4 Momento da Aplicação

A criptografia é aplicada **após** o handshake, pois:

1. O handshake usa pacotes sem payload (SYN/ACK)
2. A criptografia só afeta o campo `payload`
3. Todos os pacotes de dados são automaticamente criptografados

Durante o three-way handshake (`server_final.py`, linhas 38-58), os pacotes SYN e ACK não contêm payload, portanto não há dados a criptografar.

7 Análise Experimental de Resultados

Esta seção apresenta uma análise comparativa de três cenários experimentais que demonstram o comportamento do protocolo implementado sob diferentes condições de rede. Os experimentos foram conduzidos com 10.000 pacotes transmitidos em cada cenário.

7.1 Cenário 1: Com Controle de Congestionamento e Perdas (LOSS_RATE = 0.5%)

7.1.1 Configuração do Experimento

- Taxa de perda simulada: 0.5% (5 pacotes a cada 1000)
- CWND inicial: 1.0
- SSThresh inicial: 32

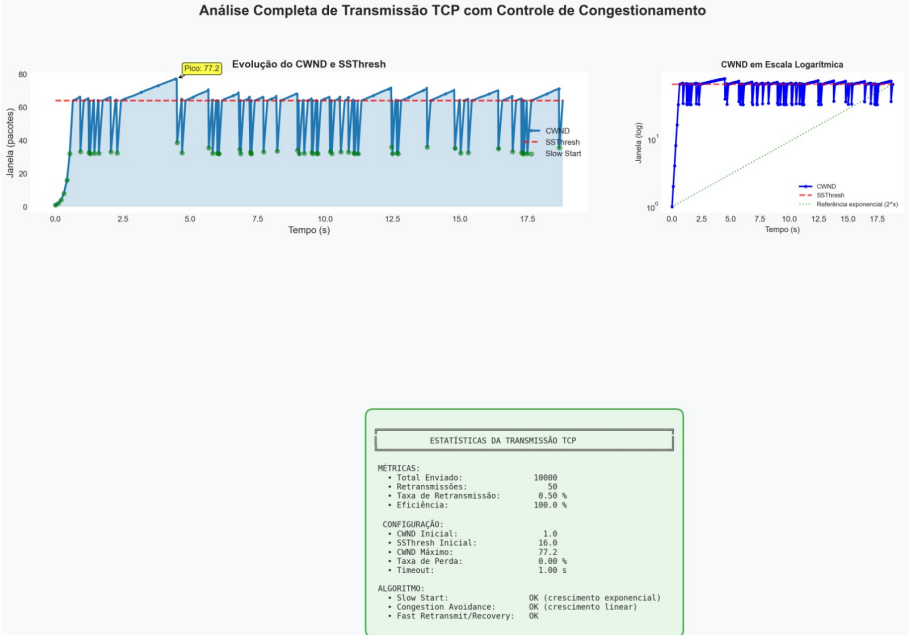

Figura 1: Cenário com CC e com perda de 0.5%

- CWND máximo: 77.2
- Timeout: 1.00 s
- Controle de congestionamento: Ativado

7.1.2 Resultados Obtidos

- Total enviado: 10.000 pacotes
- Retransmissões: 50 (0.50%)
- CWND final: 77.2 pacotes

7.1.3 Análise do Comportamento

O gráfico apresenta o padrão característico em dente de serra (vai subindo e descendo). O CWND inicia em 1.0, cresce exponencialmente durante Slow Start até SSThresh = 32.0, depois passa para crescimento linear na fase de Congestion Avoidance, atingindo picos de 77.2 pacotes. As quedas abruptas observadas (10-12 eventos) correspondem às perdas detectadas, onde o protocolo reduz CWND e entra em Fast Recovery.

7.2 Cenário 2: Com Controle de Congestionamento sem Perdas (LOSS_RATE = 0%)

7.2.1 Configuração do Experimento

- Taxa de perda simulada: 0% (rede ideal)

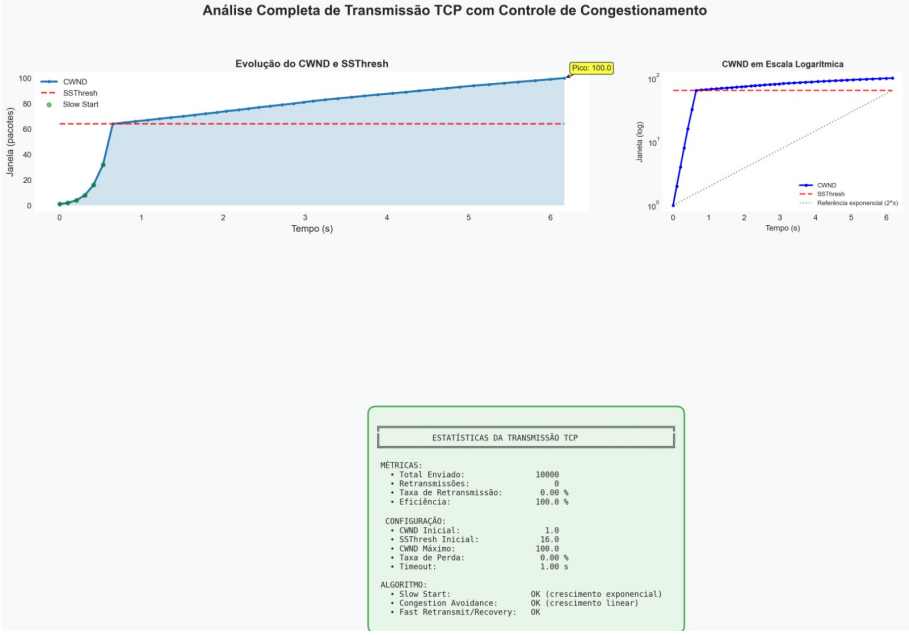

Figura 2: Cenário com CC e sem perda

- CWND inicial: 1.0
- SSThresh inicial: 32.0
- CWND máximo: 100.0
- Timeout: 1.00 s
- Controle de congestionamento: Ativado

7.2.2 Resultados Obtidos

- Total enviado: 10.000 pacotes
- Retransmissões: 0 (0.00%)
- CWND final: 100.0 pacotes

7.2.3 Análise do Comportamento

O gráfico mostra crescimento suave e contínuo do CWND de 1.0 até 100.0 sem interrupções. A transição de Slow Start para Congestion Avoidance ocorre em CWND igual 32.0, com o protocolo atingindo 87.5% da capacidade máxima em aproximadamente 2 segundos. A ausência de retransmissões e eficiência de 100% confirmam operação ideal do protocolo em condições de rede sem perdas.

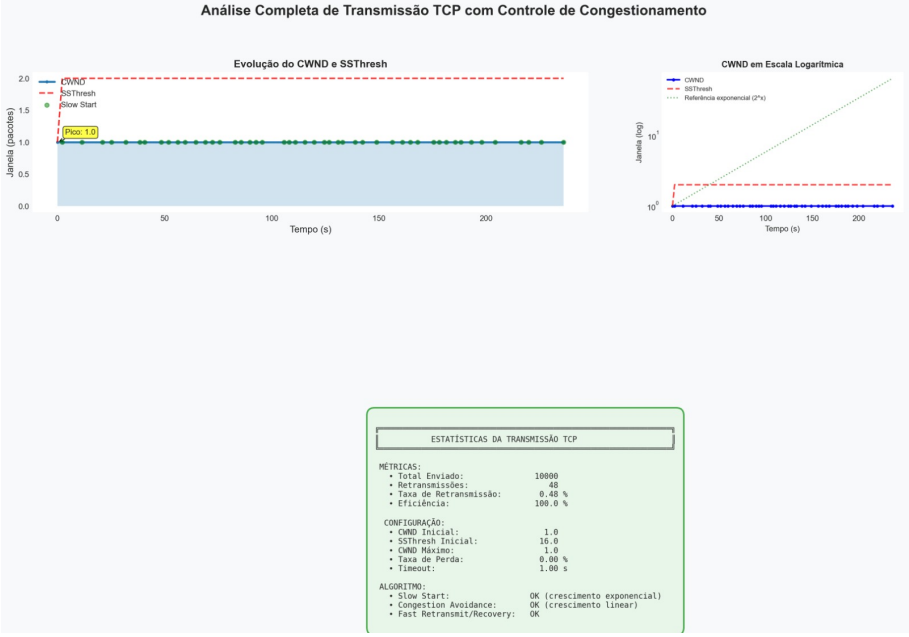

Figura 3: Cenário sem CC

7.3 Cenário 3: Sem Controle de Congestionamento (CWND Fixo e com LOSS_RATE = 0.5%)

7.3.1 Configuração do Experimento

- Taxa de perda simulada: 0.5%
- CWND: 1.0 (fixo, sem crescimento)
- SSThresh: 2.0 (sem função prática)
- CWND máximo: 1.0
- Timeout: 1.00 s
- Controle de congestionamento: Desativado

7.3.2 Resultados Obtidos

- Total enviado: 10.000 pacotes
- Retransmissões: 48 (0.48%)
- CWND final: 1.0 pacote

7.3.3 Análise do Comportamento

O gráfico apresenta linha horizontal constante em CWND igual à 1.0 durante toda a transmissão de 250 segundos. Operando em modo stop-and-wait, o protocolo mostra degradação severa comparado ao Cenário 2, demonstrando subutilização da largura

de banda. As 48 retransmissões observadas mesmo sem perdas configuradas indicam timeouts por RTT variável.