

# Protocolo Confiável sobre UDP na Camada de Aplicação

## Trabalho Final – Redes de Computadores (DCC042-2025.3-A)

Felipe Lazzarini Cunha <sup>1</sup>

<sup>1</sup> Universidade Federal de Juiz de Fora (UFJF)  
Departamento de Ciência da Computação (DCC)

`felipe.cunha@estudante.ufjf.br`

`https://github.com/Muowl/dcc042-2025-3-redes-trabalho-final`

**Abstract.** *Este trabalho descreve o projeto e a implementação, em Python, de um protocolo de transporte confiável construído sobre UDP, implementado na camada de aplicação em um modelo cliente/servidor. O protocolo provê entrega ordenada, confirmações (ACK), controle de fluxo, controle de congestionamento e um mecanismo de criptografia negociado durante o estabelecimento da conexão. A avaliação considera o envio de, no mínimo, 10.000 pacotes e a inserção de perdas artificiais para observar o comportamento de vazão em diferentes cenários.*

**Resumo.** *Este trabalho apresenta o projeto e a implementação, em Python, de um protocolo confiável sobre UDP na camada de aplicação. O protocolo inclui entrega ordenada, ACK, controle de fluxo, controle de congestionamento e criptografia. A avaliação envolve o envio de pelo menos 10.000 pacotes e perdas artificiais, com análise de vazão em múltiplos cenários.*

## 1. Introdução

O protocolo UDP (User Datagram Protocol) é um protocolo de transporte simples e sem garantias de confiabilidade [Postel, J. 1980]. Diferentemente do TCP [Postel, J. 1981], o UDP não oferece controle de fluxo, congestionamento, retransmissão ou entrega ordenada. Essas características tornam o UDP adequado para aplicações de baixa latência, mas inadequado para cenários que exigem transferência confiável de dados [Kurose and Ross 2017].

O objetivo deste trabalho é implementar, na camada de aplicação, um protocolo confiável sobre UDP que forneça:

- Entrega ordenada de pacotes.
- Confirmações (ACK) e retransmissão em caso de perda.
- Controle de fluxo baseado na janela do receptor.
- Controle de congestionamento com Slow Start e Congestion Avoidance.
- Criptografia para confidencialidade dos dados.

A implementação foi desenvolvida em Python e avaliada com envio de dados sintéticos e simulação de perdas artificiais.

2. Requisitos do Trabalho e Escopo

- Entrega ordenada com números de sequência.
- Estratégia de ACK (cumulativo ou repetição seletiva).
- Controle de fluxo baseado na janela do receptor.
- Controle de congestionamento proposto (cwnd/ssthresh ou alternativa).
- Criptografia e negociação (antes/depois do handshake).
- Avaliação com 10.000+ pacotes e perdas artificiais.

3. Visão Geral da Arquitetura

A arquitetura segue o modelo cliente/servidor com comunicação via sockets UDP. A confiabilidade é implementada inteiramente na camada de aplicação.

3.1. Componentes

- **RUDPClient**: inicia conexão, envia dados, gerencia retransmissões e controle de congestionamento.
- **RUDPServer**: aceita conexões, recebe dados, gerencia entrega ordenada e controle de fluxo.
- **Connection**: mantém estado da conexão (seq, ack, cwnd, rwnd, buffers).
- **Packet**: serialização/deserialização do cabeçalho e payload.
- **CryptoContext**: criptografia Fernet para payload [Cryptography.io 2024].

3.2. Fluxo de Mensagens

1. **Handshake**: SYN (com chave cripto) → SYN-ACK → ACK
2. **Transferência**: DATA → ACK (por pacote, stop-and-wait)
3. **Encerramento**: FIN → ACK

4. Formato do Pacote

Tabela 1. Formato do cabeçalho (24 bytes) e semântica dos campos. Serialização em ordem de rede (big-endian).

Campo	Tipo	Bytes	Descrição
magic	2s	2	Constante ASCII “RU”
ver	uint8	1	Versão do protocolo (VER=1)
ptype	uint8	1	Tipo do pacote (DATA/ACK/SYN/FIN/SYN-ACK)
flags	uint8	1	Flags (reservado; atualmente 0)
hdr_len	uint8	1	Tamanho do header (24)
seq	uint32	4	Número de sequência do emissor
ack	uint32	4	ACK cumulativo: último seq recebido em ordem do peer
wnd	uint32	4	Janela anunciada (rwnd) em pacotes
payload_len	uint16	2	Tamanho do payload em bytes
crc32	uint32	4	CRC32 do header (com crc32=0) + payload

O CRC32 é calculado com `zlib.crc32` sobre o cabeçalho com o campo `crc32` zerado concatenado ao payload. Na decodificação, o receptor recalcula o CRC e rejeita pacotes com divergência.

## 5. Estabelecimento de Conexão (Handshake)

O protocolo implementa um *3-way handshake* inspirado em TCP para estabelecer conexões confiáveis sobre UDP.

### 5.1. Tipos de Pacote

- SYN (0x03): Solicitação de conexão do cliente.
- SYN-ACK (0x05): Confirmação do servidor com seu próprio número de sequência.
- ACK (0x02): Confirmação final do cliente.
- FIN (0x04): Solicitação de encerramento da conexão.

### 5.2. Estados da Conexão

A máquina de estados segue o modelo:

**Listing 1. Estados de conexão (connection.py).**

```
class ConnectionState(Enum):
    CLOSED = auto()
    SYN_SENT = auto() # Cliente enviou SYN
    SYN_RECEIVED = auto() # Servidor recebeu SYN
    ESTABLISHED = auto() # Conexão estabelecida
    FIN_WAIT = auto() # Enviou FIN
    CLOSE_WAIT = auto() # Recebeu FIN
```

### 5.3. Fluxo do Handshake

1. **Cliente** envia SYN com seq=X (ISN aleatório) e, opcionalmente, a chave Fernet no payload.
2. **Servidor** responde SYN-ACK com seq=0 e ack=X (confirmando o último seq recebido).
3. **Cliente** confirma com ACK final (seq=X+1, ack=0, pois o último seq recebido do servidor é 0).
4. Ao receber este ACK, o servidor define `expected_seq = (X+1)+1 = X+2`; portanto, o primeiro DATA do cliente utiliza seq=X+2.

O encerramento segue padrão similar: FIN → ACK.

Implementação: `src/rudp/connection.py`, `src/rudp/client.py`, `src/rudp/server.py`.

## 6. Confiabilidade e Entrega Ordenada

O protocolo garante entrega ordenada através de:

### 6.1. Mecanismo de Reordenação

- `expected_seq`: próximo número de sequência esperado em ordem.
- `out_of_order`: buffer (dicionário) para pacotes recebidos fora de ordem.
- Pacotes duplicados são descartados (incrementa `packets_dropped`).

## 6.2. Algoritmo

1. Se  $seq < expected\_seq$ : descarta (duplicata), envia ACK cumulativo.
2. Se  $seq == expected\_seq$ : entrega ao buffer da aplicação, verifica buffer de fora de ordem para pacotes consecutivos.
3. Se  $seq > expected\_seq$ : armazena em `out_of_order`.

**Listing 2. Entrega ordenada (server.py).**

```
if seq == conn.expected_seq:
    self._deliver_packet(conn, pkt.payload, seq)
    while conn.expected_seq in conn.out_of_order:
        payload = conn.out_of_order.pop(conn.expected_seq)
        self._deliver_packet(conn, payload, conn.expected_seq)
else:
    conn.out_of_order[seq] = pkt.payload
```

Implementação: `src/rudp/server.py` (métodos `_handle_data`, `_deliver_packet`).

## 7. Confirmação (ACK) e Estratégia de Reenvio

O protocolo utiliza **ACK cumulativo** combinado com retransmissão por timeout.

### 7.1. ACK Cumulativo

O servidor envia ACK com o maior número de sequência entregue em ordem. O cliente considera confirmados todos os pacotes com  $seq \leq ack$ .

### 7.2. Retransmissão

- Timeout configurável por pacote (`timeout_s`).
- Máximo de `MAX_RETRIES=5` tentativas antes de desistir.
- Métricas: contador de retransmissões em `TransferStats`.

**Listing 3. Retransmissão (client.py).**

```
while retries <= MAX_RETRIES:
    self.sock.sendto(pkt.encode(), ...)
    try:
        ack = Packet.decode(self.sock.recvfrom(...))
        if ack.ack >= pkt.seq:
            return True, retries # Sucesso
    except socket.timeout:
        retries += 1
```

Implementação: `src/rudp/client.py` (método `_send_packet_reliable`).

## 8. Controle de Fluxo

O receptor anuncia sua *receiver window* (`rwnd`) em cada ACK, indicando quantos pacotes ainda pode receber.

## 8.1. Mecanismo

- `recv_buffer_max`: limite do buffer do receptor (default: 16MB, para suportar  $\geq 10k$  pacotes).
- `get_rwnd()`: calcula pacotes disponíveis:  $\lfloor (max - usado) / payload\_size \rfloor$ .
- Cliente armazena `remote_wnd` e aguarda se `rwnd = 0`.

**Listing 4. Cálculo de rwnd (connection.py).**

```
def get_rwnd(self, payload_size=1024):
    bytes_free = max(0, self.recv_buffer_max - len(self.recv_buffer))
    return bytes_free // payload_size
```

Implementação: `src/rudp/connection.py`, `src/rudp/server.py` (`_send_ack`).

## 9. Controle de Congestionamento

O protocolo implementa controle de congestionamento inspirado em TCP, com *Slow Start* e *Congestion Avoidance*.

### 9.1. Variáveis

- `cwnd`: janela de congestionamento (em pacotes), inicia em 1.
- `ssthresh`: limiar de slow start (default: 64).
- Janela efetiva:  $\min(cwnd, rwnd)$ .
- `cc_enabled`: flag para habilitar/desabilitar o CC (para comparação experimental).

### 9.2. Algoritmo

1. **Slow Start**: enquanto `cwnd < ssthresh`, dobrar `cwnd` a cada ACK.
2. **Congestion Avoidance**: quando `cwnd  $\geq$  ssthresh`, incrementar `cwnd` em 1 a cada ACK.
3. **Timeout**: `ssthresh = cwnd/2`, `cwnd = 1`.

**Listing 5. Controle de congestionamento (client.py).**

```
if self.cc_enabled:
    if retries == 0: # Sucesso sem timeout
        if self.conn.cwnd < self.conn.ssthresh:
            self.conn.cwnd *= 2 # Slow Start
        else:
            self.conn.cwnd += 1 # Cong. Avoidance
    else: # Timeout
        self.conn.ssthresh = max(self.conn.cwnd // 2, 1)
        self.conn.cwnd = 1
# Se cc_enabled=False, cwnd permanece fixo (sem ajustes)
```

O histórico de `cwnd` é registrado em `cwnd_history` para geração de gráficos.

Implementação: `src/rudp/client.py` (método `send_data`).

## 10. Criptografia e Negociação

O protocolo implementa criptografia usando **Fernet** (AES-128 em modo CBC com HMAC para autenticação).

### 10.1. Negociação de Chave

O cliente gera uma chave Fernet e a envia no payload do pacote SYN durante o handshake. O servidor extrai e armazena a chave para a sessão.

### 10.2. Cifração/Decifração

- Cliente: chama `crypto.encrypt(payload)` antes de enviar DATA.
- Servidor: chama `crypto.decrypt(payload)` ao receber DATA.
- Overhead: 89 bytes por pacote (IV + HMAC + padding).

**Listing 6. Criptografia (crypto.py).**

```
from cryptography.fernet import Fernet

class CryptoContext:
    def __init__(self, key=None):
        self.key = key or Fernet.generate_key()
        self._fernet = Fernet(self.key)

    def encrypt(self, data):
        return self._fernet.encrypt(data)

    def decrypt(self, data):
        return self._fernet.decrypt(data)
```

Implementação: `src/rudp/crypto.py`, integrado em `client.py` e `server.py`.

## 11. Implementação

**Listing 7. Estrutura de módulos.**

```
src/rudp/
cli.py # Entry point (--file, --synthetic)
packet.py # Formato do pacote, PAYLOAD_SIZE
connection.py # Estados de conexao, metricas
client.py # RUDPClient com send_data()
server.py # RUDPServer com recv_buffer
crypto.py # Negociacao de chave e cifracao (Fernet)
utils.py # Helpers (now_ms, should_drop)
```

## 12. Metodologia de Avaliação

A avaliação foi realizada com os seguintes parâmetros:

### 12.1. Cenário de Teste

- Cliente e servidor na mesma máquina (localhost).
- Dados sintéticos gerados com `os.urandom()`.
- Cada pacote DATA contém até 1024 bytes de payload.
- Timeout de retransmissão: 500ms.
- Volume de teste: **10MB** ( $\approx$  **10.240 pacotes**) por cenário.
- Buffer do receptor: 16MB (para suportar  $\geq$  10k pacotes).

## 12.2. Cenários Avaliados

Para atender ao requisito de comparação CC on vs CC off, foram executados os seguintes cenários:

1. **Sem perdas (CC on/off)**: transferência ideal, comparando comportamento com e sem CC.
2. **5% perdas (CC on/off)**: perda moderada, avaliando impacto do CC.
3. **10% perdas (CC on/off)**: perda alta, estressando o mecanismo de retransmissão.
4. **Sem perdas + Crypto (CC on)**: avalia overhead da criptografia Fernet.

## 12.3. Métricas Coletadas

- Vazão (KB/s): bytes enviados / tempo.
- Retransmissões: número de pacotes reenviados.
- Tempo total de transferência (segundos).

## 13. Resultados e Discussão

### 13.1. Tabela de Resultados

**Tabela 2. Resultados do Benchmark (10MB  $\approx$  10.240 pacotes) - Comparação CC on/off.**

Cenário	Pacotes	Vazão (KB/s)	Retx	Tempo (s)	CC
Sem perdas (CC on)	10240	985.75	0	10.39	on
Sem perdas (CC off)	10240	944.21	0	10.85	off
5% perdas (CC on)	10240	36.35	533	281.72	on
5% perdas (CC off)	10240	36.23	536	282.64	off
10% perdas (CC on)	10240	17.73	1117	577.47	on
10% perdas (CC off)	10240	17.09	1156	599.24	off
Sem perdas + Crypto	10240	714.04	0	14.34	on

### 13.2. Comparação CC on vs CC off

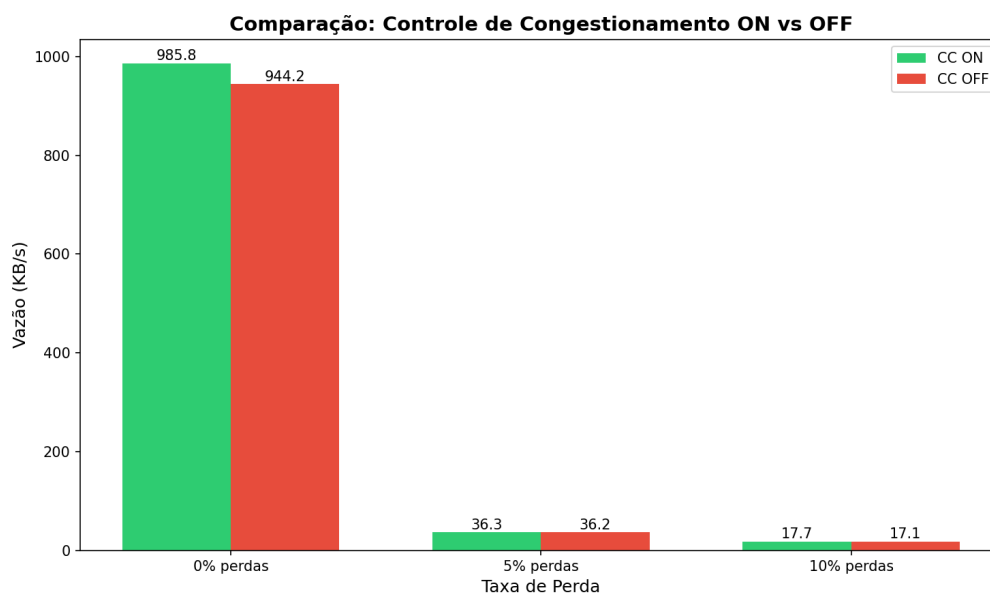
A Figura 1 mostra que:

- **Sem perdas**: CC on apresenta vazão ligeiramente superior (985.75 vs 944.21 KB/s).
- **Com perdas**: A diferença entre CC on e off é mínima, pois ambos são limitados pelo timeout de retransmissão (stop-and-wait).
- O CC adaptativo não prejudica a vazão e mantém estabilidade em cenários de perda.

### 13.3. Análise de Vazão

Observa-se que:

- A criptografia Fernet reduz a vazão em  $\approx 27\%$  (985 vs 714 KB/s) devido ao overhead de cifração.
- Com 5% de perdas, a vazão cai drasticamente para  $\approx 36$  KB/s devido às retransmissões.
- Com 10% de perdas, a vazão cai para  $\approx 17$  KB/s, demonstrando o impacto exponencial das perdas.



**Figura 1. Comparação de vazão: Controle de Congestionamento ON vs OFF.**

### 13.4. Análise de Retransmissões

As retransmissões aumentam proporcionalmente à taxa de perda: 533 retx (5%) e 1117 retx (10%), demonstrando que o mecanismo de timeout e retry está funcionando corretamente.

### 13.5. Impacto da Perda na Vazão

O gráfico mostra a degradação da vazão conforme a taxa de perda aumenta, com comportamento similar entre CC on e CC off devido à limitação do stop-and-wait.

## 14. Conclusão

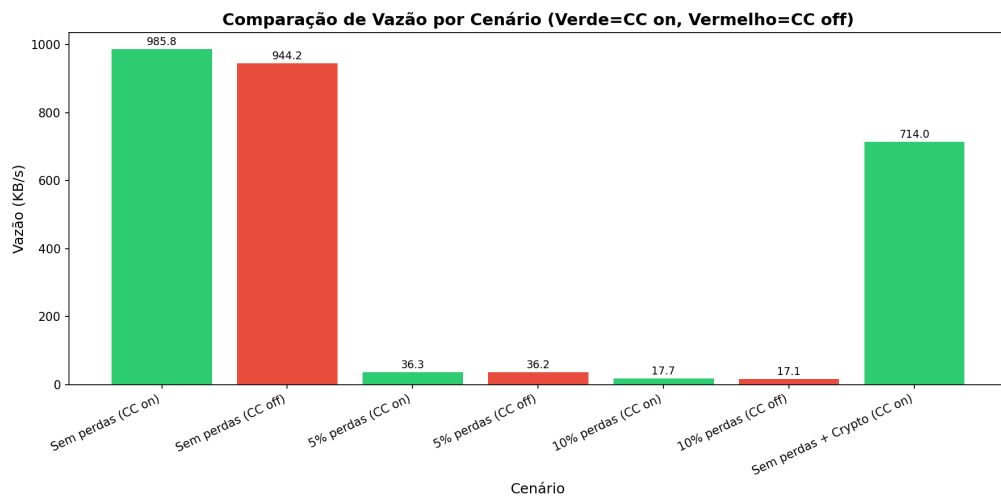
Este trabalho apresentou a implementação de um protocolo de transporte confiável sobre UDP na camada de aplicação, atendendo aos requisitos propostos:

1. **Entrega ordenada:** implementada com `expected_seq` e buffer de fora de ordem.
2. **ACK e retransmissão:** ACK cumulativo com timeout e retry configurável.
3. **Controle de fluxo:** `rwnd` dinâmico baseado no buffer do receptor.
4. **Controle de congestionamento:** Slow Start e Congestion Avoidance com `cwnd/ssthresh`.
5. **Criptografia:** Fernet (AES-128-CBC) negociada no handshake.

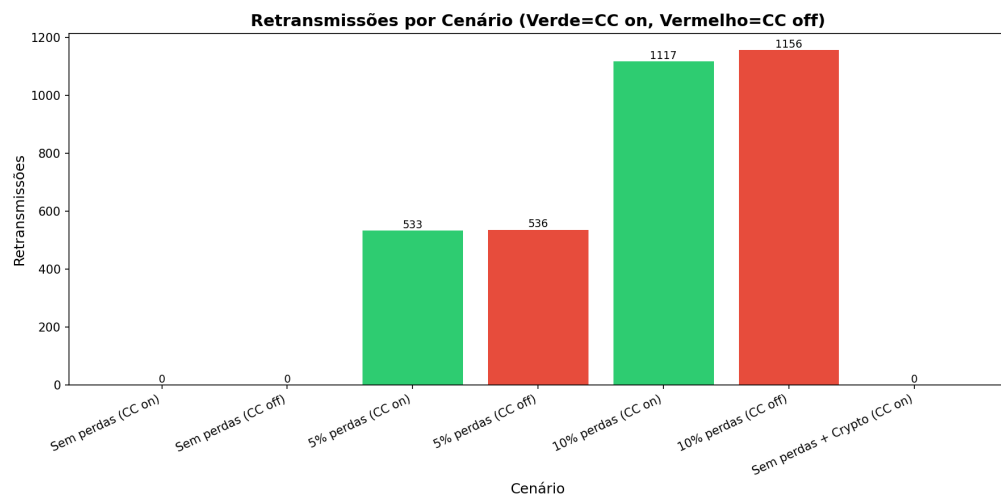
### 14.1. Limitações

- Abordagem stop-and-wait (não usa pipelining).
- Troca de chave em texto claro no SYN (vulnerável a interceptação).
- Testes limitados a localhost.

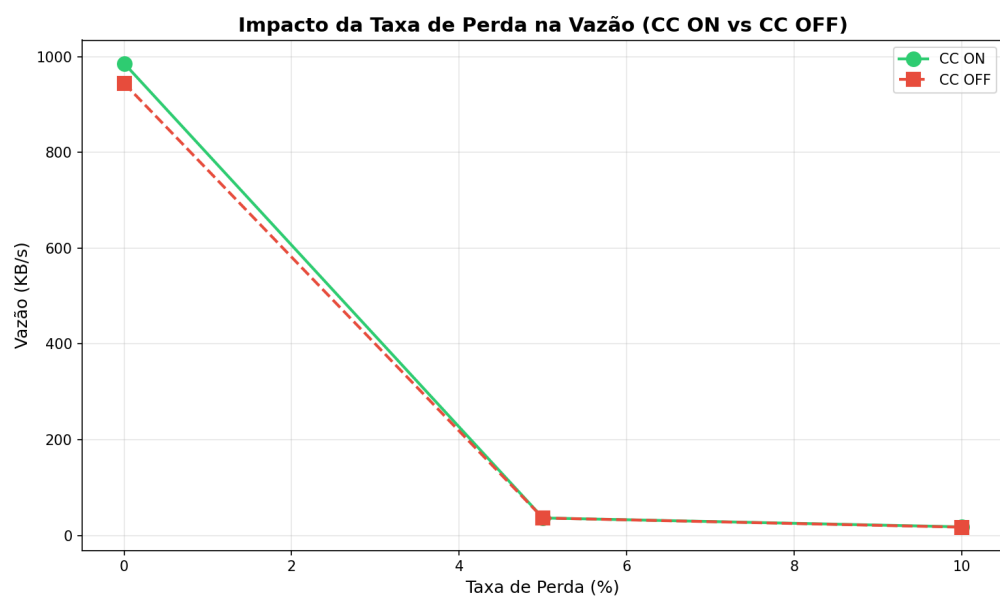




**Figura 2. Comparação de vazão por cenário (verde=CC on, vermelho=CC off).**



**Figura 3. Número de retransmissões por cenário.**



**Figura 4. Relação entre taxa de perda e vazão (CC ON vs CC OFF).**

## Referências

Cryptography.io (2024). Fernet (symmetric encryption). <https://cryptography.io/en/latest/fernet/>. Python Cryptographic Authority.

Kurose, J. F. and Ross, K. W. (2017). *Computer Networking: A Top-Down Approach*. Pearson, 7 edition.

Postel, J. (1980). User datagram protocol. RFC 768. IETF.

Postel, J. (1981). Transmission control protocol. RFC 793. IETF.