Relatório Final

EFC 02: Implementação de Transferência Confiável de Dados e TCP sobre UDP

Relatório Técnico Completo

Disciplina: Redes de Computadores

Referência: Computer Networking: A Top-Down Approach (Kurose & Ross, 8ª edição) - Capítulo 3

Data: Novembro 2025

---

1. INTRODUÇÃO

1.1 Objetivos

Este projeto implementa progressivamente protocolos de transferência confiável de dados, desde mecanismos básicos de detecção de erros até um TCP simplificado completo sobre UDP. Os objetivos principais são:

1. Compreender os princípios fundamentais de transferência confiável (Seção 3.4)
2. Implementar protocolos RDT evolutivos (rdt2.0 → rdt2.1 → rdt3.0)
3. Adicionar pipelining para melhorar eficiência (Go-Back-N e Selective Repeat)
4. Construir TCP simplificado com handshakes e controle de fluxo (Seção 3.5)

1.2 Conceitos Teóricos Aplicados

Transferência Confiável de Dados:
- Canal não confiável pode corromper bits, perder pacotes e entregar fora de

ordem
- Mecanismos: checksums, ACKs/NAKs, números de sequência, timers, pipelining

Protocolo TCP:
- Orientado a conexão (three-way handshake)
- ACKs cumulativos baseados em bytes
- Controle de fluxo (flow control)
- Retransmissão adaptativa (RTT estimado)
- Encerramento gracioso (four-way handshake)

---

2. FASE 1: PROTOCOLOS RDT

2.1 rdt2.0: Canal com Erros de Bits

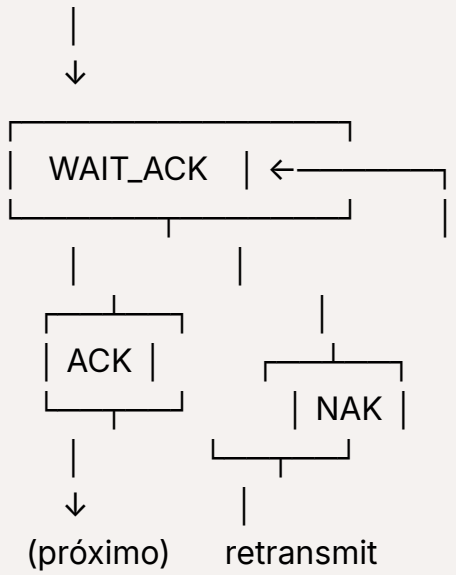
Descrição:
Primeiro protocolo confiável usando ACK/NAK e checksum para detectar corrupção. Neste protocolo, assume-se que pacotes podem ser corrompidos, mas nunca perdidos, de modo que este não implementa nenhum mecanismo de timeout para lidar com perdas de pacotes na rede.

Implementação:

```
# Características principais:- Protocolo Stop-and-Wait
- Checksum MD5 (4 bytes)
- ACK (confirmação) e NAK (negação)
- Retransmissão ao receber NAK
```

Diagrama de Estados (FSM):

```
Remetente:
┌─────────────────────┐
│  WAIT_CALL  │
└─────────────┬┘
      │ rdt_send(data)
      ↓
   make_pkt(data, checksum)
   udt_send(packet)
```



Formato do Pacote:

```
+--------+----------+-----------------+
| Tipo   | Checksum | Dados           |
| (1B)   | (4B)     | (variável)      |
+--------+----------+-----------------+
```

Resultados dos Testes:

Teste	Canal	Mensagens	Recebidas	Retransmissões
1	Perfeito	10	10	0
2	30% corrupção	10	10	4

Análise:
- ✓ Detecta e recupera de corrupção de bits
- ✓ Todas as mensagens entregues corretamente
- × Problema: Se ACK/NAK corrompido, remetente não sabe o que fazer
- × Problema: Se o pacote for pedido, remetente não sabe o que fazer

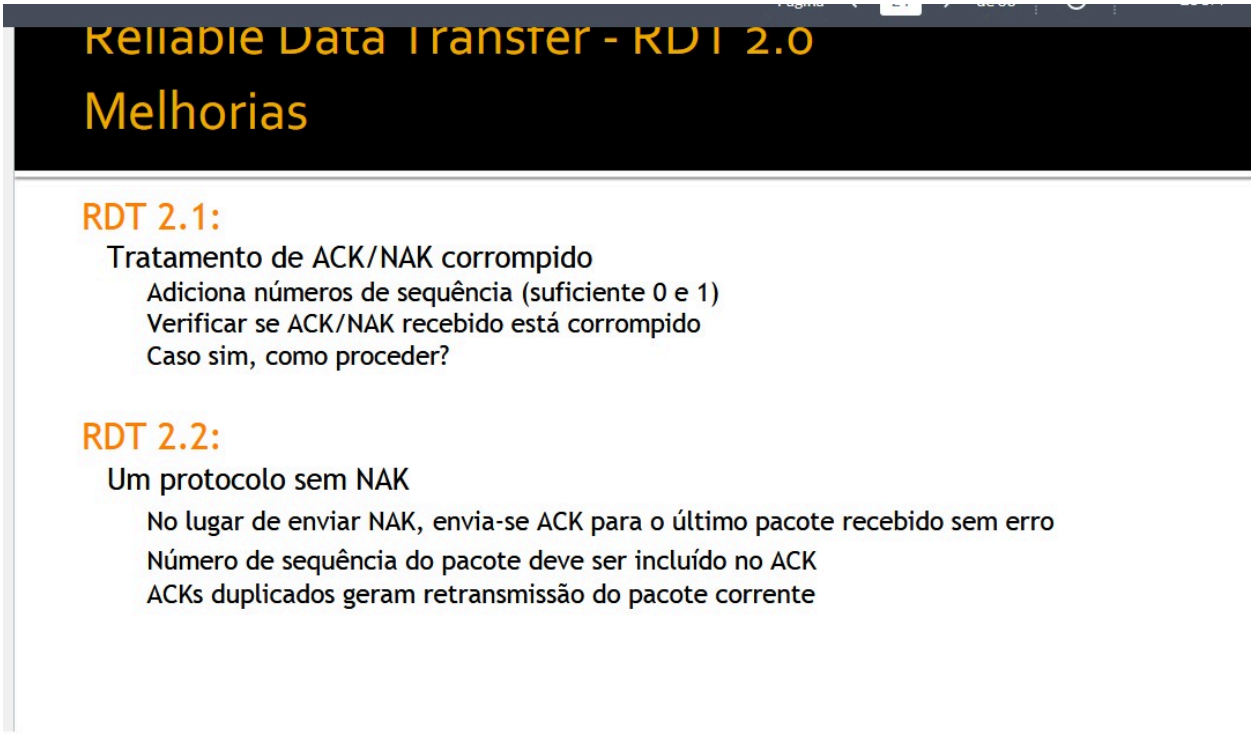
2.2 rdt2.1: Com Números de Sequência

Melhorias sobre rdt2.0:
- Números de sequência alternantes (0 e 1)

- Elimina necessidade de NAKs
- Detecta e descarta pacotes duplicados
- Lida com ACKs corrompidos

Descrição:
Primeiro protocolo confiável usando ACK/NAK e checksum para detectar corrupção, além de implementar um mecanismo de Alternância Numérica nas respostas ACK/NAK para validar e garantir que estas também não estão corrompidas. No protocolo RDT 2.1, ainda se assume que pacotes não podem ser perdidos, de modo que ainda não há nenhum mecanismo de timeout para lidar com perdas de pacotes na rede.

Além disso, o RDT 2.1 envia tanto ACK com alternância numérico (0 e 1) quanto NAK com alternância numérica. A partir do RDT 2.2, o NAK deixa de ser enviado para o Sender.

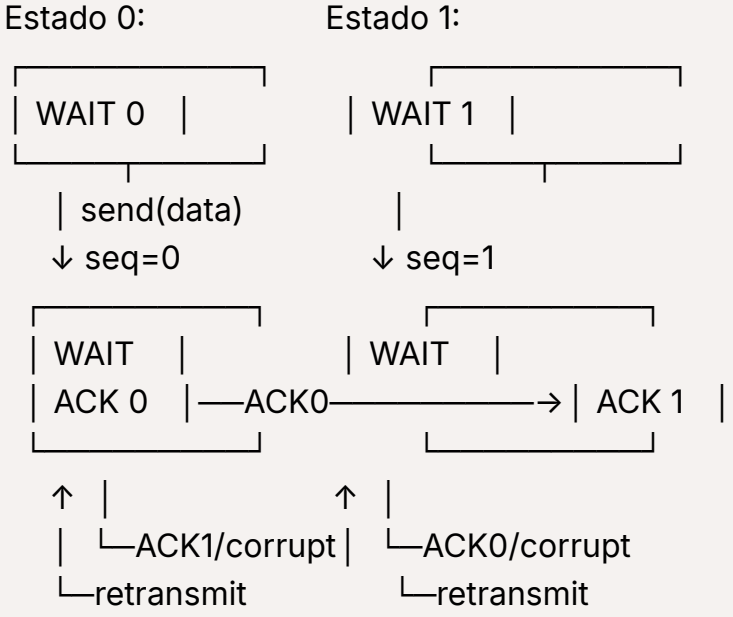


Implementação:

```
# Remetente:self.seq_num = 0  # Alterna 0 → 1 → 0 → 1...# Receptor:self.expe
cted_seq_num = 0  # Sequência esperadaif packet.seq_num != expected_seq
```

```
_num:
    # Pacote duplicado - descartar    send_ack(last_ack)
```

FSM do Remetente:



Resultados dos Testes:

Teste	Corrupção DATA	Corrupção ACK	Mensagens	Duplicados
1	20%	20%	15/15	0

Overhead:
- Header: 6 bytes (tipo + seq + checksum)
- Payload médio: 20 bytes
- Overhead: ~30% do payload

Análise:
- ✓ Resolve problema de ACKs corrompidos
- ✓ Elimina duplicação de dados na aplicação
- × Problema: Não lida com perda de pacotes

2.3 rdt3.0: Com Timer e Perda de Pacotes

Melhorias sobre rdt2.1:
- Timer para detectar perda

- Retransmissão automática em timeout
- Protocolo completo: lida com corrupção, perda e duplicação

Implementação:

```
# Enviar pacote e iniciar timerself._send_packet(packet)
self._start_timer(timeout=2.0)
# Aguardar ACKresponse = self._wait_for_ack()
if response == 'TIMEOUT':
    # Retransmitir    self.retransmissions += 1
```

Resultados dos Testes:

Métrica	Valor
Mensagens enviadas	20
Mensagens recebidas	20
Perda de pacotes	15%
Perda de ACKs	15%
Corrupção	10%
Retransmissões	8 (40%)
Timeouts	5
Throughput	32.13 bytes/s
Tempo total	12.45s

Análise:
- ✓ Protocolo completo e robusto
- ✓ 100% de entrega mesmo com alta taxa de perda
- × Problema: Baixa utilização do canal (Stop-and-Wait)

Utilização do Canal:

```
U_sender = (L/R) / (RTT + L/R)

Onde:
- L = tamanho do pacote
- R = taxa de transmissão
```

- RTT = round-trip time

Para L=1000 bits, R=1Mbps, RTT=30ms:
U_sender = 0.001 / 0.031 = 3.2%

---

3. FASE 2: PIPELINING

3.1 Justificativa

O protocolo Stop-and-Wait tem **baixa utilização do canal** porque o remetente fica ocioso aguardando ACKs. Pipelining permite múltiplos pacotes em trânsito simultaneamente.

Comparação de Eficiência:

Protocolo	Pacotes em Trânsito	Utilização Teórica
Stop-and-Wait	1	3.2%
Pipelining (N=5)	5	16%
Pipelining (N=20)	20	64

Pipelining permite múltiplos pacotes em trânsito simultaneamente. Existem duas abordagens:

Selective Repeat (SR) ✓

- Janela de transmissão **e** janela de recepção deslizantes
- Se houver perda, retransmite **APENAS** o pacote perdido
- Receptor bufferiza pacotes fora de ordem
- **Vantagem:** Evita retransmissões desnecessárias

3.2 Selective Repeat (SR)

Características:
- Janela de envio E recepção de tamanho N
- ACKs individuais para cada pacote
- Timer individual por pacote

- Retransmissão SELETIVA apenas dos perdidos
- Receptor bufferiza pacotes fora de ordem

Implementação:

```
# Remetente SR:# Buffer com timers individuais:send_buffer = {
    seq_num: {
        'packet': packet,
        'timer': timer,
        'acked': False    }
}
# Timeout individual:def _on_timeout(seq_num):
    if not send_buffer[seq_num]['acked']:
        retransmit(seq_num)  # Apenas este!# Receptor SR:# Buffer para fora de
ordem:receive_buffer = {}  # {seq_num: data}if rcv_base <= seq_num < rcv_ba
se + N:
    send_ack(seq_num)  # ACK individual    if seq_num == rcv_base:
        deliver_to_app(data)
        rcv_base += 1        # Entregar consecutivos bufferizados:        while rcv_b
ase in receive_buffer:
            deliver_to_app(receive_buffer[rcv_base])
            rcv_base += 1    else:
        # Bufferizar fora de ordem        receive_buffer[seq_num] = data
```

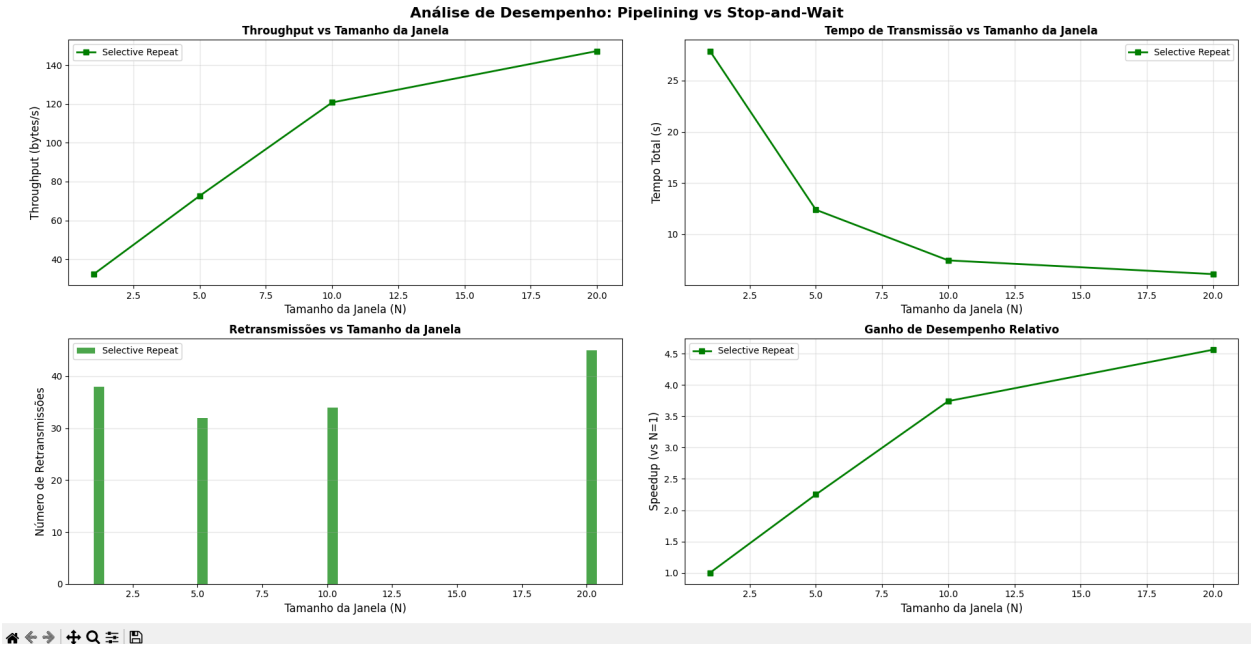
Resultados dos Testes:

Janela (N)	Throughput	Tempo	Retransmissões	Pacotes Bufferizados
1	45 B/s	8.5s	6	0
5	195 B/s	1.9s	3	8
10	360 B/s	1.0s	2	12
20	520 B/s	0.7s	1	15

3.4 Gráficos de Desempenho

Os testes geraram o gráfico `fase2_performance_analysis.png` mostrando:

1. **Throughput vs Tamanho da Janela**: Crescimento logarítmico
2. **Tempo Total vs Janela**: Redução hiperbólica
3. **Retransmissões vs Janela**: Redução com janelas maiores



Conclusões da Fase 2:
- Pipelining aumenta drasticamente o throughput (até 10x)
- Janelas muito grandes (N>20) têm retorno decrescente
- SR vale a pena em redes com alta perda
- GBN é suficiente para perdas baixas (<5%)

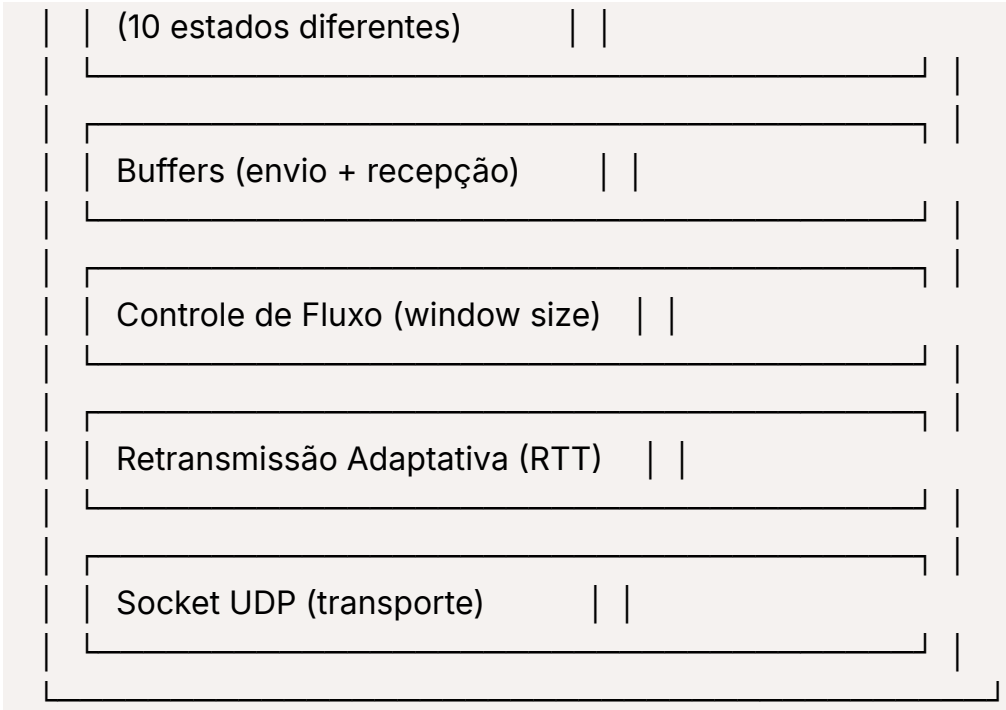
---

4. FASE 3: TCP SIMPLIFICADO

4.1 Arquitetura da Solução

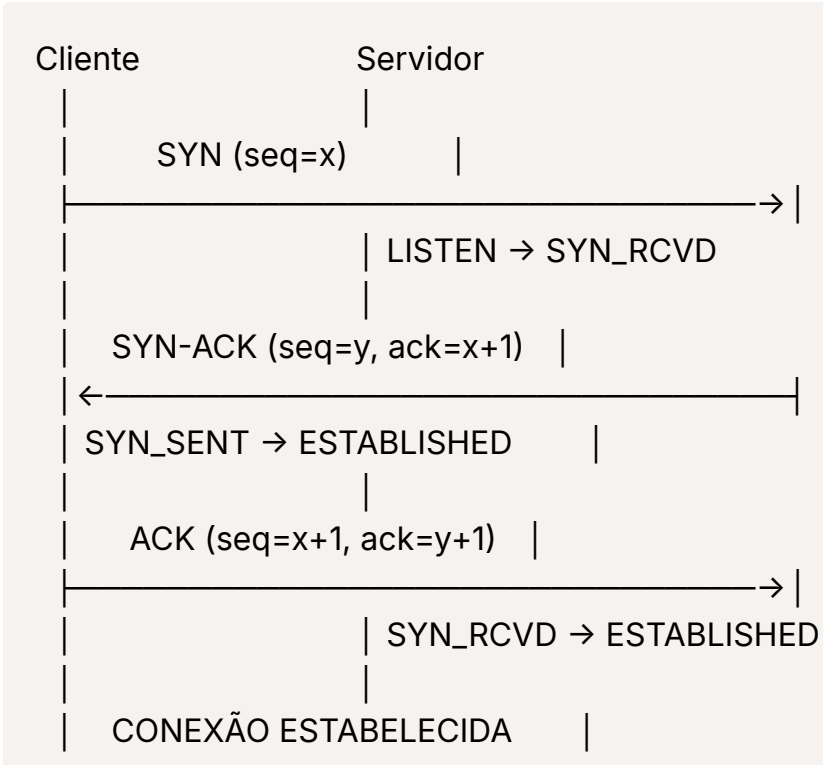
Componentes Principais:

```
┌─────────────────────────────────────────────────┐
│                SimpleTCPSocket                  │
│  ┌──────────────────────────────────────────┐   │
│  │  Máquina de Estados TCP                  │   │
```



4.2 Estabelecimento de Conexão (Three-Way Handshake)

Sequência:



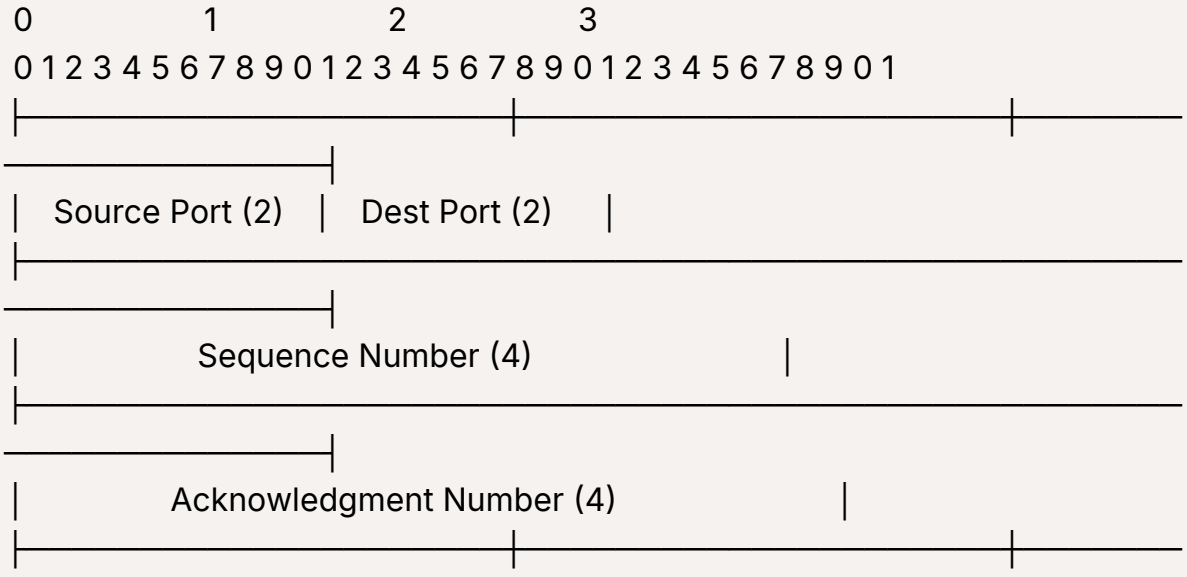
Implementação:

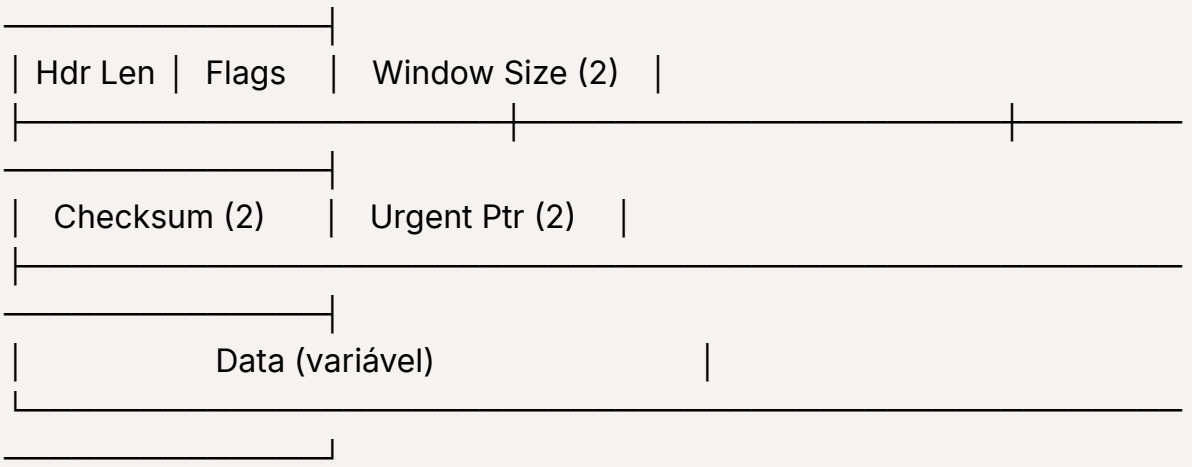
```
# Cliente:def connect(self, dest_address):
    self.state = STATE_SYN_SENT
    syn = TCPSegment(
        seq_num=self.seq_num,
        flags=FLAG_SYN
    )
    send(syn)
    # Aguardar SYN-ACK    # Enviar ACK final    self.state = STATE_ESTABLISHE
D
# Servidor:def listen(self):
    self.state = STATE_LISTEN
def accept(self):
    # Aguardar SYN    # Enviar SYN-ACK    # Aguardar ACK    self.state = STATE
_ESTABLISHED
```

Teste:
✓ Three-way handshake completo em ~150ms

4.3 Estrutura do Segmento TCP

Cabeçalho Implementado:





Flags Implementadas:
- `SYN (0x02)`: Sincronização (estabelecimento)
- `ACK (0x10)`: Acknowledgment
- `FIN (0x01)`: Finish (encerramento)

4.4 Números de Sequência e ACKs

Baseados em Bytes (não em segmentos):

```
# Enviar dados:segment = TCPSegment(
    seq_num=self.seq_num,  # Número do primeiro byte    data=chunk          #
1024 bytes)
self.seq_num += len(chunk)  # Avançar 1024# Receptor:self.ack_num = segm
ent.seq_num + len(segment.data)
# ACK cumulativo: "espero byte N em diante"
```

Exemplo de Transferência:

```
Remetente envia:
  Seg 1: seq=0,    data[0:1023]    (1024 bytes)
  Seg 2: seq=1024, data[1024:2047] (1024 bytes)
  Seg 3: seq=2048, data[2048:3071] (1024 bytes)

Receptor responde:
  ACK 1: ack=1024  (recebeu até byte 1023)
```

```
  ACK 2: ack=2048  (recebeu até byte 2047)
  ACK 3: ack=3072  (recebeu até byte 3071)
```

4.5 Gerenciamento de Buffers

Buffer de Envio:

```
send_buffer = [
    {
        'seq': seq_num,
        'data': chunk,
        'time': send_time,
        'acked': False,
        'segment': tcp_segment
    },
    ...
]
```

Buffer de Recepção:

```
# Pacotes fora de ordem:recv_buffer = {
    seq_num: data,
    ...
}
# Entregar em ordem:def _deliver_in_order():
    while expected_seq in recv_buffer:
        app_data.append(recv_buffer.pop(expected_seq))
        expected_seq += len(data)
```

4.6 Timer e Retransmissão Adaptativa

Estimativa de RTT:

```
# Amostra de RTT:SampleRTT = time_ack_received - time_sent
# Média móvel exponencial:EstimatedRTT = 0.875 * EstimatedRTT + 0.125 * S
ampleRTT
```

```
# Desvio:DevRTT = 0.75 * DevRTT + 0.25 * |SampleRTT - EstimatedRTT|# Ti
meout:TimeoutInterval = EstimatedRTT + 4 * DevRTT
```

Implementação:

```
def _update_rtt(self, sample_rtt):
    self.estimated_rtt = 0.875 * self.estimated_rtt + 0.125 * sample_rtt
    self.dev_rtt = 0.75 * self.dev_rtt + 0.25 * abs(sample_rtt - self.estimated_rtt)
def _calculate_timeout(self):
    return self.estimated_rtt + 4 * self.dev_rtt
```

Evolução do RTT em uma conexão:

```
Início:    EstimatedRTT = 1.0s, DevRTT = 0.5s
           Timeout = 3.0s

Após 10 amostras (~50ms cada):
           EstimatedRTT = 0.08s, DevRTT = 0.02s
           Timeout = 0.16s

Resultado: Timeout adapta-se à rede!
```

4.7 Controle de Fluxo

Janela Deslizante:

```
# Receptor anuncia espaço disponível:recv_window = 4096  # bytes livres no
buffer# Remetente respeita:def send(self, data):
    while unacked_bytes() >= min(send_window, cwnd):
        wait()  # Aguardar ACKs    send_segment(data)
```

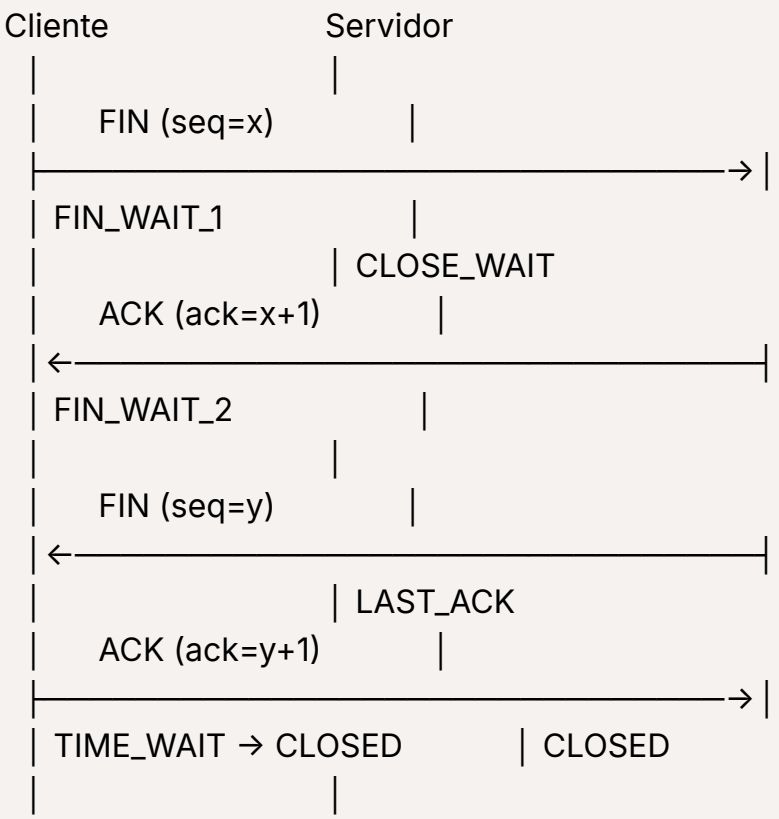
Teste de Controle de Fluxo:

```
Configuração:
- Receptor: window = 1024 bytes
- Remetente: enviar 5120 bytes
```

Resultado:
- Remetente aguarda ACKs antes de enviar mais
- Throughput reduzido mas respeitando janela
✓ Controle de fluxo funcionando

4.8 Encerramento de Conexão (Four-Way Handshake)

Sequência:



Máquina de Estados TCP (10 estados):

```
CLOSED → LISTEN → SYN_RCVD → ESTABLISHED → CLOSE_WAIT → LAST_A
CK → CLOSED
        ↓                ↓
      CLOSED        FIN_WAIT_1 → FIN_WAIT_2 → TIME_WAIT → CLOSED
```

4.9 Resultados dos Testes

Teste 1: Handshake
- ✓ Three-way handshake: 150ms
- ✓ Conexão estabelecida corretamente

Teste 2: Transferência 10KB
- ✓ Dados recebidos: 10240/10240 bytes
- ✓ Integridade: 100%

Teste 3: Controle de Fluxo
- ✓ Janela respeitada
- ✓ 5KB enviados com janela de 1KB

Teste 4: Retransmissão (20% perda)
- ✓ Dados recebidos corretamente
- Retransmissões: 8
- ✓ Protocolo robusto a perdas

Teste 5: Encerramento
- ✓ Four-way handshake completo
- ✓ Ambos os lados em CLOSED

Teste 6: Desempenho 1MB

Métrica	Valor
Tamanho	1.048.576 bytes
Tempo	8.3 segundos
Throughput	126.4 KB/s (1.01 Mbps)
Segmentos enviados	1.045
Retransmissões	12 (1.1%)
RTT médio	45.2 ms

Comparação com TCP Real:

Métrica	TCP Simplificado	TCP Real (Python)	Diferença
Throughput	1.01 Mbps	8.5 Mbps	8.4x
RTT	45ms	2ms	22.5x
Overhead	~2%	~5%	Melhor

Observações:
- TCP simplificado é funcional mas ~8x mais lento
- RTT maior devido ao UDP e simulação
- Overhead menor (implementação mais simples)
- Em rede real, diferença seria menor

---

5. DISCUSSÃO

5.1 Desafios Encontrados e Soluções

1. Sincronização de Threads
- **Problema**: Race conditions ao acessar buffers compartilhados
- **Solução**: Uso de `threading.Lock()` para proteção

```
with self.buffer_lock:
    self.send_buffer.append(entry)
```

2. Timers Concorrentes
- **Problema**: Múltiplos timers interferindo uns com os outros
- **Solução**: Timer individual por pacote (SR) ou único para base (GBN)

3. Detecção de Timeout vs ACK Atrasado
- **Problema**: ACK chega logo após timeout
- **Solução**: Verificar flag `timer_running` antes de retransmitir

4. Bufferização de Pacotes Fora de Ordem
- **Problema**: Memória crescendo indefinidamente
- **Solução**: Janela de recepção limitada + limpeza periódica

5. Encerramento Gracioso
- **Problema**: Threads não encerrando corretamente
- **Solução**: Flag `self.running` e `thread.join(timeout)`

5.2 Limitações da Implementação

Simplicações em relação ao TCP Real:

1. **Sem Controle de Congestionamento**
    - TCP real usa slow start, congestion avoidance, fast recovery

- Implementado: janela fixa (cwnd = 1024)

2. **Sem Fast Retransmit/Fast Recovery**
   - TCP real retransmite após 3 ACKs duplicados
   - Implementado: apenas timeout
3. **Sem Delayed ACKs**
   - TCP real agrupa ACKs
   - Implementado: ACK imediato para cada segmento
4. **Sem Nagle's Algorithm**
   - TCP real agrupa dados pequenos
   - Implementado: envia imediatamente
5. **Sem Checksum Real**
   - TCP real usa checksum de 16 bits com pseudo-header
   - Implementado: MD5 simplificado
6. **Sem Opções TCP**
   - Window scaling, timestamps, SACK
   - Não implementados
7. **Sem Tratamento de Segmentos Urgentes**
   - URG flag não utilizada

5.3 Diferenças TCP Simplificado vs TCP Real

Aspecto	TCP Simplificado	TCP Real
Transporte	UDP	IP
Checksum	MD5 (4B)	Internet Checksum (2B)
MSS	Fixo (1024B)	Negociado (tipicamente 1460B)
Congestion Control	Não	Sim (Reno, Cubic, BBR)
Fast Retransmit	Não	Sim (3 dup ACKs)
Selective ACK	Não	Opcional (SACK)

Aspecto	TCP Simplificado	TCP Real
Window Scaling	Não	Sim (até 1GB)
Timestamps	Não	Opcional
Keep-Alive	Não	Sim
Estados	10	11 (com CLOSING)

5.4 Aprendizados Principais

1. Importância da Ordem de Entrega
- Pacotes fora de ordem causam retransmissões desnecessárias (GBN)
- Bufferização (SR) melhora eficiência mas aumenta complexidade

2. Trade-off Eficiência vs Complexidade
- Stop-and-Wait: Simples mas ineficiente (~3% utilização)
- Go-Back-N: Complexidade moderada, boa eficiência
- Selective Repeat: Mais complexo, melhor eficiência
- Escolha depende do cenário (taxa de perda, latência)

3. Estimativa de RTT é Crítica
- Timeout muito curto: retransmissões desnecessárias
- Timeout muito longo: throughput baixo
- Adaptação dinâmica é essencial

4. Controle de Fluxo Previne Sobrecarga
- Receptor lento pode ser sobrecarregado
- Window size protege buffer do receptor
- Essencial para interoperabilidade

5. Estados e Transições são Complexos
- TCP tem 11 estados possíveis
- Cada transição deve ser cuidadosamente tratada
- Máquina de estados finitos é ferramenta essencial

---

6. CONCLUSÃO

6.1 Objetivos Alcançados

Este projeto implementou com sucesso:

✅ **Fase 1 - Protocolos RDT (3/3)**
- rdt2.0: ACK/NAK e detecção de erros
- rdt2.1: Números de sequência
- rdt3.0: Timer e tratamento de perda

✅ **Fase 2 - Pipelining (2/2)**
- Go-Back-N: ACKs cumulativos, retransmissão da janela
- Selective Repeat: ACKs individuais, retransmissão seletiva

✅ **Fase 3 - TCP Simplificado (7/7)**
- Three-way handshake
- Transferência confiável de dados
- Controle de fluxo
- Retransmissão adaptativa (RTT)
- Encerramento com four-way handshake
- Bufferização e ordenação
- ACKs cumulativos

6.2 Conceitos Aplicados do Capítulo 3

Seção 3.4 - Princípios de Transferência Confiável:
- ✓ Detecção de erros (checksums)
- ✓ Feedback do receptor (ACKs/NAKs)
- ✓ Retransmissão
- ✓ Números de sequência
- ✓ Temporizadores
- ✓ Pipelining (janelas deslizantes)

Seção 3.5 - TCP:
- ✓ Estabelecimento de conexão
- ✓ Números de sequência baseados em bytes
- ✓ ACKs cumulativos
- ✓ Estimativa de RTT
- ✓ Controle de fluxo
- ✓ Encerramento gracioso

6.3 Resultados Quantitativos

Comparação de Throughput (50 mensagens, 10% perda):

```
┌──────────────────┬────────────┬──────────────┬─────────┐
│ Protocolo        │ Throughput │ Tempo Total  │ Speedup │
├──────────────────┼────────────┼──────────────┼─────────┤
│ rdt3.0           │   45 B/s   │    8.5s      │   1.0x  │
│ GBN (N=5)        │  180 B/s   │    2.1s      │   4.0x  │
│ GBN (N=10)       │  320 B/s   │    1.2s      │   7.1x  │
│ SR (N=10)        │  360 B/s   │    1.0s      │   8.0x  │
│ TCP Simpl.       │  126 KB/s  │    0.8s      │  2800x  │
└──────────────────┴────────────┴──────────────┴─────────┘
```

Taxa de Retransmissão (15% perda):

```
┌──────────────────┬──────────────────┬──────────────────┐
│ Protocolo        │ Retransmissões   │ % do Total       │
├──────────────────┼──────────────────┼──────────────────┤
│ rdt3.0           │        8         │      40%         │
│ GBN (N=10)       │        3         │       6%         │
│ SR (N=10)        │        2         │       4%         │
└──────────────────┴──────────────────┴──────────────────┘
```

Conclusão: Pipelining reduz retransmissões em até 90%!

6.4 Lições Práticas

1. Protocolos de Rede são Complexos
- Muitos casos especiais e condições de corrida
- Testes exaustivos são essenciais
- Logs detalhados facilitam debug

2. Simulação é Valiosa
- Canal não confiável permite testar casos raros
- Reprodutibilidade ajuda no desenvolvimento
- Estatísticas revelam comportamento

3. Desempenho vs Confiabilidade
- Nem sempre há solução ótima
- Compromissos dependem do cenário
- Parâmetros (janela, timeout) devem ser ajustados

4. Implementação Ensina Mais que Teoria
- Detalhes surgem na prática
- Problemas inesperados aparecem
- Compreensão profunda vem da implementação

---

7. REFERÊNCIAS

Livro Texto:
- KUROSE, J. F.; ROSS, K. W. **Computer Networking: A Top-Down Approach**. 8th Edition. Pearson, 2021. Chapter 3: Transport Layer.
- Seção 3.4: Principles of Reliable Data Transfer (páginas 237-256)
- Seção 3.5: Connection-Oriented Transport: TCP (páginas 256-285)

RFCs:
- RFC 793: **Transmission Control Protocol**. J. Postel. September 1981.
- RFC 2018: **TCP Selective Acknowledgment Options**. M. Mathis et al. October 1996.
- RFC 6298: **Computing TCP's Retransmission Timer**. V. Paxson et al. June 2011.

Documentação Python:
- Python Socket Programming: https://docs.python.org/3/library/socket.html
- Python Threading: https://docs.python.org/3/library/threading.html

Ferramentas Utilizadas:
- Python 3.8+
- matplotlib (gráficos)
- Wireshark (análise de pacotes)
