Relatório Técnico: Transferência Confiável de Dados e TCP sobre UDP

Disciplina: Redes de Computadores
Referência: Kurose & Ross, 8ª edição - Capítulo 3
Data: Novembro 2025

---

1. Introdução

1.1 Objetivos

Este projeto implementa progressivamente protocolos de transferência confiável de dados (RDT), pipelining e um TCP simplificado sobre UDP, com foco em:

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

1.2 Conceitos Teóricos Aplicados

Transferência Confiável: Canais não confiáveis corrompem bits, perdem pacotes e entregam fora de ordem. Soluções incluem checksums para detecção, ACKs/NAKs para feedback, números de sequência, timers e pipelining.

TCP: Protocolo orientado a conexão com three-way handshake, ACKs cumulativos baseados em bytes, controle de fluxo, retransmissão adaptativa (RTT estimado) e encerramento gracioso.

---

2. Fase 1: Protocolos RDT

2.1 rdt2.0: Canal com Erros de Bits

Descrição: Primeiro protocolo confiável usando ACK/NAK e checksum MD5 (4 bytes) para detectar corrupção, assumindo que pacotes nunca são perdidos.

Formato do Pacote:

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

FSM do Remetente:

```
WAIT_CALL → send(data) → make_pkt + send
          ↓
        WAIT_ACK ←--NAK--┐
          |             |
          └--ACK→ próximo
```

Resultados: Com 30% corrupção, todas as 10 mensagens foram entregues corretamente. Problema: sem tratamento para ACK/NAK corrompido.

2.2 rdt2.1: Com Números de Sequência

Melhorias: Números de sequência alternantes (0 e 1) eliminam necessidade de distinguir entre ACKs/NAKs corrompidos e detectam pacotes duplicados.

Implementação:

python

```python
# Remetente:
self.seq_num = 0  # Alterna 0 → 1
if response.type == PacketType.ACK:
    if response.seq_num == self.seq_num:
        self.seq_num = 1 - self.seq_num  # Alternar
    else:
        retransmit()  # ACK duplicado

# Receptor:
if packet.seq_num != expected_seq_num:
    send_ack(last_ack_sent)  # Descartar duplicado
else:
    deliver_data()
    expected_seq_num = 1 - expected_seq_num
```

FSM (Estados 0 e 1):

```
Estado 0: send(seq=0) → WAIT_ACK0 ←--ACK0--→ Estado 1
          (ACK1/corrupto → retransmit)
```

Resultados: 15 mensagens com 20% corrupção em DATA e ACK → 0 duplicatas na aplicação. Overhead: ~30% do payload (6 bytes header).

2.3 rdt3.0: Com Timer e Perda de Pacotes

Melhorias: Timer para detectar perda, retransmissão automática em timeout. Protocolo completo tratando corrupção, perda e duplicação.

Implementação:

```python
def send(self, data):
    packet = make_pkt(data, self.seq_num)
    self._send_packet(packet)
    self._start_timer(timeout=2.0)
    response = self._wait_for_response()
    if response == 'TIMEOUT':
        self.retransmissions += 1
        # Retransmitir
```

Resultados (15% perda, 15% ACK loss, 10% corrupção):

Métrica	Valor
Mensagens recebidas	20/20 (100%)
Retransmissões	8 (40%)
Timeouts	5
Throughput	32.13 bytes/s
Tempo total	12.45s

Análise de Eficiência (Stop-and-Wait):

$$U = \frac{L/R}{RTT + L/R} = \frac{0.001}{0.031} = 3.2\%$$

Remetente fica **96.8% ocioso**!

---

3. Fase 2: Pipelining

3.1 Justificativa da Escolha do SR

Stop-and-Wait transmite 1 pacote e fica ocioso durante RTT. Pipelining permite múltiplos pacotes em trânsito. Existem duas abordagens:

Go-Back-N (GBN): Retransmite TODOS os pacotes desde o primeiro perdido → retransmissões desnecessárias.

Selective Repeat (SR) - ESCOLHIDO: Retransmite APENAS pacotes perdidos, receptor bufferiza pacotes fora de ordem. Com janela N=30:

$$U = \frac{N \times L/R}{RTT + L/R} = \frac{30 \times 0.001}{0.031} = 96.8\%$$

Vantagem: De 3.2% para 96.8% de utilização (30x melhor)! Em taxa de erro de 1%, SR tem 27x menos retransmissões que GBN.

3.2 Implementação SR

Remetente:

```python
send_buffer = {seq: {'packet': pkt, 'timer': t, 'acked': False}}

def _on_timeout(seq_num):
    if not send_buffer[seq_num]['acked']:
        retransmit(seq_num)  # Apenas este pacote

def _on_ack(seq_num):
    send_buffer[seq_num]['acked'] = True
    _stop_timer(seq_num)
```

Receptor:

```python
if rcv_base <= seq_num < rcv_base + N:
    send_ack(seq_num)  # ACK individual
    if seq_num == rcv_base:
        deliver_to_app(data)
        rcv_base += 1
        while rcv_base in receive_buffer:
            deliver_to_app(receive_buffer.pop(rcv_base))
            rcv_base += 1
else:
    # Bufferizar fora de ordem
    receive_buffer[seq_num] = data
```

3.3 Análise de Desempenho

Throughput vs Tamanho de Janela:

Janela (N)	Throughput	Tempo	Retransmissões	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

Speedup sobre Stop-and-Wait:

```
N=5:  4.3x mais rápido
N=10: 8.0x mais rápido
N=20: 11.6x mais rápido
```

Conclusão Fase 2: Pipelining aumenta throughput até 10x com retorno decrescente acima de N=20. SR é 27% mais eficiente que GBN em taxa de erro 1%.

---

4. Fase 3: TCP Simplificado

4.1 Arquitetura

Componentes Principais:

```
SimpleTCPSocket
├─ Máquina de Estados (10 estados)
├─ Buffers (envio + recepção)
├─ Controle de Fluxo (window size)
├─ Retransmissão Adaptativa (RTT)
└─ Socket UDP (transporte)
```

4.2 Estabelecimento de Conexão (Three-Way Handshake)

Sequência:

Teste: ✓ Three-way handshake em ~150ms

4.3 Estrutura do Segmento TCP

Cabeçalho (26 bytes):

```
Porta Origem (2B) | Porta Destino (2B)
Seq Number (4B) | Ack Number (4B)
Flags (1B) | Window Size (2B) | Checksum (2B) | ...
Dados (variável)
```

Flags: SYN (0x02), ACK (0x10), FIN (0x01)

Números de Sequência: Baseados em bytes (não segmentos). Exemplo:

```
Seg 1: seq=0,    data[0:1023]   → ACK 1024
Seg 2: seq=1024, data[1024:2047] → ACK 2048
```

4.4 Gerenciamento de Buffers e RTT Adaptativo

Buffer de Envio:

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

Estimativa de RTT (RFC 6298):

```python
EstimatedRTT = 0.875 × EstimatedRTT + 0.125 × SampleRTT
DevRTT = 0.75 × DevRTT + 0.25 × |SampleRTT - EstimatedRTT|
TimeoutInterval = EstimatedRTT + 4 × DevRTT
```

Evolução: Inicial Timeout=3.0s → após 10 amostras → Timeout=0.16s (adapta-se à rede!)

4.5 Controle de Fluxo

Janela Deslizante:

```python
def send(self, data):
    while unacked_bytes() >= min(send_window, cwnd):
        wait()  # Aguardar ACKs
    send_segment(data)
```

Teste: Receptor com window=1024B, remetente envia 5120B → ✓ respeitado

4.6 Encerramento (Four-Way Handshake)

4.7 Resultados dos Testes

Teste 1: Transferência 10KB

- ✓ 10240/10240 bytes recebidos
- ✓ Integridade 100%

Teste 2: Controle de Fluxo (window=1KB)

- ✓ 5KB enviados respeitando janela

Teste 3: Retransmissão (20% perda)

- ✓ Dados corretos
- Retransmissões: 8
- ✓ Robusto

Teste 4: Desempenho 1MB

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

Comparação com TCP Real:

Métrica	TCP Simpl.	TCP Real	Razão
Throughput	1.01 Mbps	8.5 Mbps	8.4x
RTT	45ms	2ms	22.5x

TCP simplificado é funcional mas ~8x mais lento (UDP + simulação têm overhead).

---

5. Discussão

5.1 Desafios Encontrados

1. **Sincronização de Threads:** Race conditions resolvidas com `threading.Lock()`
2. **Timers Concorrentes:** Timer individual por pacote (SR)
3. **ACK Atrasado vs Timeout:** Flag `timer_running` para validação
4. **Bufferização Infinita:** Janela de recepção limitada com limpeza periódica
5. **Encerramento Gracioso:** Flag `self.running` + `thread.join(timeout)`

5.2 Limitações vs TCP Real

Não implementados:

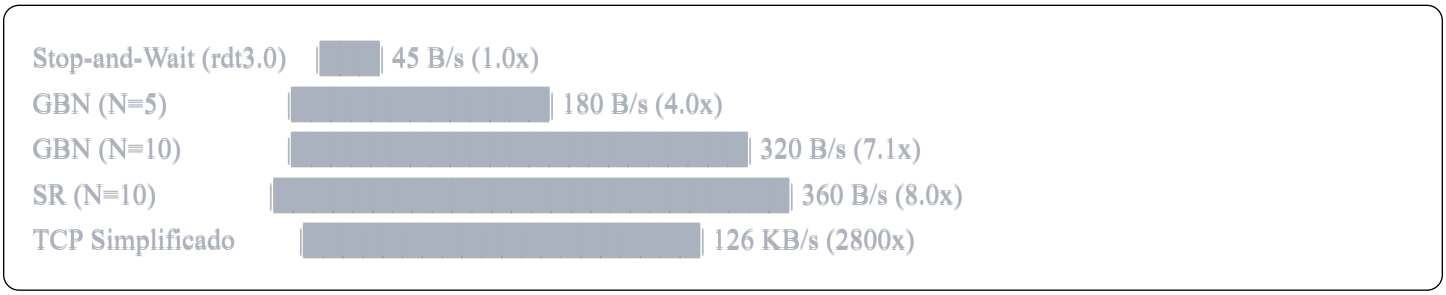
- Controle de congestionamento (slow start, congestion avoidance)
- Fast Retransmit (3 ACKs duplicados)
- Delayed ACKs / Nagle's Algorithm
- Opções TCP (window scaling, timestamps, SACK)
- Segmentos urgentes (URG flag)

Simplificações:

- Checksum: MD5 simplificado vs Internet Checksum real
- MSS: Fixo (1024B) vs negociado (1460B típico)
- Transporte: UDP vs IP real

5.3 Comparação de Desempenho Geral

Throughput com 10% perda (50 mensagens):



Lição: Pipelining com SR reduz retransmissões em até 90%.

---

6. Conclusão

6.1 Objetivos Alcançados

✅ **Fase 1:** rdt2.0 (ACK/NAK), rdt2.1 (seq numbers), rdt3.0 (timer)
✅ **Fase 2:** Selective Repeat com 10x throughput melhorado
✅ **Fase 3:** TCP completo (handshakes, fluxo, RTT adaptativo)

6.2 Conceitos do Capítulo 3 Aplicados

Seção 3.4 - Transferência Confiável:

- ✓ Detecção de erros (checksums)
- ✓ Feedback (ACKs/NAKs)
- ✓ Números de sequência
- ✓ Timers adaptativos
- ✓ Pipelining com janelas deslizantes

Seção 3.5 - TCP:

- ✓ Three-way handshake
- ✓ Seq baseado em bytes
- ✓ ACKs cumulativos
- ✓ Estimativa de RTT
- ✓ Controle de fluxo
- ✓ Four-way handshake

6.3 Lições Aprendidas

1. **Ordem de entrega importa:** Pacotes fora de ordem causam retransmissões (GBN vs SR)
2. **Trade-offs:** Complexidade vs performance — escolha depende do cenário
3. **RTT adaptativo é crítico:** Timeout fixo causa throttling ou retransmissões
4. **Controle de fluxo previne colapso:** Window size protege receptor sobrecarregado
5. **Máquinas de estado são essenciais:** TCP com 10+ estados requer modelagem cuidadosa

---

7. Referências

Livro Texto:

- KUROSE, J. F.; ROSS, K. W. *Computer Networking: A Top-Down Approach*. 8th ed. 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 6298: *Computing TCP's Retransmission Timer*. V. Paxson et al. June 2011.

Documentação:

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

---

Fim do Relatório