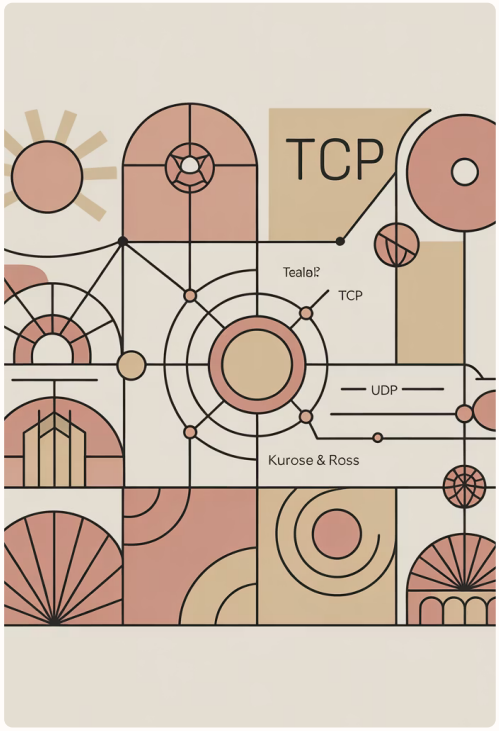
EFC 02: Implementação de Transferência Confiável de Dados e TCP sobre UDP

Redes de Computadores

Referência: Capítulo 3 - Camada de Transporte (Kurose & Ross, 8ª edição)

Prof. Douglas Abreu



1. OBJETIVOS DA ATIVIDADE

Esta atividade tem como objetivos:

1. Compreender os princípios fundamentais da transferência confiável de dados apresentados na Seção 3.4 do livro
2. Implementar progressivamente mecanismos de confiabilidade sobre um canal não confiável
3. Construir uma versão simplificada do TCP sobre UDP, aplicando os conceitos da Seção 3.5
4. Experimentar na prática os desafios de implementar controle de erros, retransmissão e controle de fluxo

2. CONTEXTO TEÓRICO

2.1 Transferência Confiável de Dados (Seção 3.4)

Conforme estudado no Capítulo 3, a camada de transporte deve fornecer um serviço de transferência confiável de dados sobre um canal que pode:

- **Corromper bits** nos pacotes (erros de transmissão)
- **Perder pacotes** completamente
- **Entregar pacotes fora de ordem**

O livro apresenta uma evolução incremental de protocolos (Seção 3.4.1):

- **rdt1.0**: Canal perfeitamente confiável (caso trivial)
- **rdt2.0**: Canal com erros de bits (introduz ACK/NAK)
- **rdt2.1**: Adiciona números de sequência para lidar com ACKs/NAKs corrompidos
- **rdt2.2**: Remove NAKs, usa apenas ACKs duplicados
- **rdt3.0**: Adiciona timer para lidar com perda de pacotes

2.2 TCP - Transmission Control Protocol (Seção 3.5)

O TCP é um protocolo orientado a conexão que fornece:

- Transferência confiável de dados usando os princípios do rdt
- Controle de fluxo (flow control)
- Controle de congestionamento (congestion control)
- Estabelecimento e encerramento de conexão (three-way handshake)

Características principais do TCP (Seções 3.5.2 a 3.5.4):

- Números de sequência baseados em bytes (não em segmentos)
- ACKs cumulativos
- Retransmissão baseada em timeout
- Gerenciamento de buffers de envio e recepção

3. ESTRUTURA DA ATIVIDADE

A atividade está dividida em **3 fases progressivas**. Cada fase adiciona complexidade e funcionalidades sobre a anterior.

FASE 1: IMPLEMENTAÇÃO DOS

PROTOCOLOS RDT BÁSICOS

Objetivo da Fase 1

Implementar versões incrementais do protocolo de transferência confiável de dados, seguindo a progressão apresentada na Seção 3.4.1 do livro.

3.1.1 Tarefa 1A: Implementar rdt2.0 (Canal com Erros de Bits)

Referência: Seção 3.4.1, Figura 3.10

Descrição: Implemente um protocolo simples de transferência de dados que funcione sobre UDP, incorporando:

No Lado Remetente (Sender):

1. Aceitar dados da aplicação
2. Criar um pacote com:
   - Dados
   - Checksum (pode usar checksum simples ou hash MD5)
3. Enviar via UDP
4. Aguardar ACK ou NAK do receptor
5. Se receber NAK: retransmitir o pacote
6. Se receber ACK: aceitar novos dados

No Lado Receptor (Receiver):

1. Receber pacote via UDP
2. Verificar integridade usando checksum
3. Se corrupto: enviar NAK
4. Se correto:
   - Entregar dados para aplicação
   - Enviar ACK

Protocolo Stop-and-Wait: O remetente deve aguardar ACK/NAK antes de enviar o próximo pacote.

Formato do Pacote (sugestão):

```
+----------------+----------------+
| Tipo (1 byte)  | Checksum (4)   |
+----------------+----------------+
| Dados (variável)                |
+---------------------------------+
Tipo: 0=DATA, 1=ACK, 2=NAK
```

Testes Obrigatórios:

1. Transmitir uma sequência de 10 mensagens com canal perfeito
2. Introduzir corrupção artificial de bits (inverter bits aleatórios) em 30% dos pacotes
3. Verificar se todas as mensagens chegam corretamente ao destino
4. Registrar quantas retransmissões ocorreram

3.1.2 Tarefa 1B: Implementar rdt2.1 (com Números de Sequência)

Referência: Seção 3.4.1, Figuras 3.11 e 3.12

Problema a Resolver: No rdt2.0, se um ACK/NAK for corrompido, o remetente não sabe se deve retransmitir. A solução é adicionar números de sequência aos pacotes.

Modificações Necessárias:

No Remetente:

1. Adicionar campo de número de sequência (0 ou 1 - protocolo alternante)
2. Numerar pacotes alternadamente: 0, 1, 0, 1, ...
3. Aguardar ACK com número de sequência correto
4. Se receber ACK corrompido ou com número incorreto: retransmitir

No Receptor:

1. Verificar número de sequência do pacote recebido
2. Se for o esperado e não corrompido:
   - Entregar à aplicação
   - Enviar ACK com número de sequência
   - Alternar número esperado
3. Se for duplicado (número incorreto):
   - Descartar dados
   - Reenviar ACK do pacote anterior

Formato do Pacote Atualizado:

```
+-------+-------+----------+
| Tipo  | SeqNum| Checksum |
+-------+-------+----------+
| Dados (variável)         |
+--------------------------+
SeqNum: 0 ou 1
```

Testes Obrigatórios:

1. Corromper 20% dos pacotes DATA
2. Corromper 20% dos ACKs
3. Verificar se não há duplicação de dados na aplicação receptora
4. Medir overhead (quantos bytes extras por mensagem útil)

3.1.3 Tarefa 1C: Implementar rdt3.0 (com Timer e Perda de Pacotes)

Referência: Seção 3.4.1, Figuras 3.15 e 3.16

Problema a Resolver: Pacotes ou ACKs podem ser perdidos completamente. A solução é adicionar um timer no remetente.

Modificações Necessárias:

No Remetente:

1. Iniciar um timer após enviar cada pacote
2. Se timeout ocorrer antes de receber ACK:
    - Retransmitir o pacote
    - Reiniciar timer
3. Ao receber ACK correto:
    - Cancelar timer
    - Prosseguir para próximo pacote

No Receptor:

- Nenhuma mudança necessária em relação ao rdt2.1

Configuração do Timer:

- Usar um timeout inicial de 2 segundos
- Implementar usando threads ou asyncio em Python

Testes Obrigatórios:

1. Simular perda de 15% dos pacotes DATA (descartar aleatoriamente antes de enviar)
2. Simular perda de 15% dos ACKs
3. Simular atraso variável (50-500ms) na rede
4. Verificar se todas as mensagens são entregues corretamente
5. Medir:
    - Taxa de retransmissão
    - Throughput efetivo (bytes úteis / tempo total)

FASE 2: IMPLEMENTAÇÃO DE PIPELINING (Go-Back-N OU Selective Repeat)

Objetivo da Fase 2

Melhorar a eficiência implementando pipelining, permitindo múltiplos pacotes em trânsito simultaneamente.

3.2.1 Escolha do Protocolo

Você deve escolher **UM** dos protocolos abaixo para implementar:

Opção A: Go-Back-N (GBN)

Seção 3.4.3, Figuras 3.19 e 3.20

Opção B: Selective Repeat (SR)

Seção 3.4.4, Figuras 3.23 e 3.24

Dica: O GBN é mais simples de implementar, mas o SR é mais eficiente em redes com perdas.

3.2.2 Implementação Go-Back-N (se escolher esta opção)

Referência: Seção 3.4.3

Características do GBN:

- Janela de envio de tamanho N (sugestão: N=5)
- Remetente pode ter até N pacotes não confirmados em trânsito
- ACKs cumulativos: ACK(n) confirma todos os pacotes até n
- Retransmissão de todos os pacotes da janela em caso de timeout

No Remetente:

1. Manter variáveis:
   - base: número de sequência do pacote mais antigo não confirmado
   - nextseqnum: próximo número de sequência disponível
   - N: tamanho da janela
2. Se nextseqnum < base + N:
   - Pode enviar novos pacotes
3. Manter **um único timer** para o pacote mais antigo não confirmado
4. Ao receber ACK(n):
   - Mover base para n + 1
   - Se base == nextseqnum: parar timer
   - Senão: reiniciar timer
5. Em timeout:
   - Retransmitir **todos** os pacotes de base até nextseqnum - 1

No Receptor:

1. Manter expectedseqnum
2. Se receber pacote com número correto:
   - Entregar à aplicação
   - Enviar ACK(expectedseqnum)
   - Incrementar expectedseqnum
3. Se receber pacote fora de ordem:
   - **Descartar pacote**
   - Reenviar ACK do último pacote entregue em ordem

Formato do Pacote:

```
+-------+----------+----------+
| Tipo | SeqNum | Checksum |
| | (4 bytes)| (4 bytes)|
+-------+----------+----------+
| Dados (variável) |
+--------------------------+
SeqNum: número de 0 a 2^32–1
```

3.2.3 Implementação Selective Repeat (se escolher esta opção)

Referência: Seção 3.4.4

Características do SR:

- Janela de envio e recepção de tamanho N
- ACKs individuais para cada pacote
- Retransmissão seletiva apenas dos pacotes perdidos
- Receptor bufferiza pacotes fora de ordem

No Remetente:

1. Manter variáveis semelhantes ao GBN
2. Manter **um timer individual** para cada pacote na janela
3. Ao receber ACK(n):
    - Marcar pacote n como confirmado
    - Se n for o base: mover janela para frente até próximo pacote não confirmado
4. Em timeout do pacote n:
    - Retransmitir **apenas** o pacote n

No Receptor:

1. Manter janela de recepção [rcv_base, rcv_base + N - 1]
2. Manter buffer para armazenar pacotes fora de ordem
3. Se receber pacote dentro da janela:
    - Enviar ACK individual
    - Bufferizar se fora de ordem
    - Se for rcv_base: entregar este e todos os consecutivos bufferizados, avançar janela
4. Se receber pacote fora da janela mas já confirmado:
    - Reenviar ACK (pode ter sido perdido)

3.2.4 Testes Obrigatórios para Fase 2

Para ambas as implementações (GBN ou SR):

1 Teste de Eficiência

- Transferir 1MB de dados
- Comparar tempo com rdt3.0 (stop-and-wait)
- Calcular utilização do canal

2 Teste com Perdas

- Taxa de perda de 10%
- Verificar se todas as mensagens chegam
- Contar retransmissões

3 Teste de Ordenação (só para SR)

- Verificar se pacotes fora de ordem são bufferizados corretamente

4 Análise de Desempenho

- Variar tamanho da janela (N = 1, 5, 10, 20)
- Plotar gráfico: Throughput x Tamanho da Janela

FASE 3: IMPLEMENTAÇÃO DE TCP SIMPLIFICADO SOBRE UDP

Objetivo da Fase 3

Integrar os conceitos anteriores e adicionar funcionalidades do TCP real.

3.3.1 Funcionalidades Mínimas do TCP Simplificado

Referência: Seção 3.5

Seu TCP simplificado deve incluir:

1. Estabelecimento de Conexão (Three-Way Handshake) - Seção 3.5.6

```
Cliente                    Servidor
   |                          |
   |-------- SYN ------------>|
   |                          |
   |<----- SYN-ACK -----------|
   |                          |
   |-------- ACK ------------>|
   |                          |
   |    CONEXÃO ESTABELECIDA  |
```

2. Estrutura do Segmento TCP - Seção 3.5.2

Implemente os seguintes campos do cabeçalho TCP (Figura 3.29):

```
+------------------+------------------+
| Source Port (2)  | Dest Port (2)    |
```

```
+-------------------+-------------------+
| Sequence Number (4 bytes)             |
+---------------------------------------+
| Acknowledgment Number (4 bytes)       |
+---------------------------------------+
| Header | Flags    | Window Size (2)   |
| Len(1) | (1 byte) |                   |
+---------------------------------------+
| Checksum (2)      | Urgent Ptr (2)    |
+---------------------------------------+
| Dados (variável)                      |
+---------------------------------------+
Flags: [URG|ACK|PSH|RST|SYN|FIN] (use apenas SYN, ACK e FIN)
```

3. Números de Sequência e ACKs - Seção 3.5.2

- **Número de sequência**: número do primeiro byte no segmento
- **ACK cumulativo**: próximo byte esperado
- Exemplo (Figura 3.30):
  - Segmento 1: seq=0, dados bytes 0-999
  - Segmento 2: seq=1000, dados bytes 1000-1999
  - ACK recebido: ack=2000 (espera byte 2000 em diante)

4. Gerenciamento de Buffers - Seção 3.5.1

- Buffer de envio: armazena dados até serem confirmados
- Buffer de recepção: armazena dados recebidos até serem lidos pela aplicação

5. Timer e Retransmissão - Seção 3.5.3

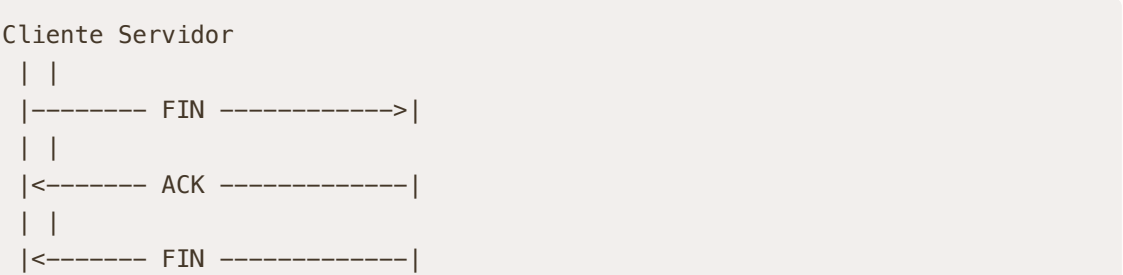
Implementar estimativa de RTT e timeout adaptativo:

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

6. Controle de Fluxo - Seção 3.5.5

- Usar campo Window Size para indicar espaço livre no buffer de recepção
- Remetente não deve exceder a janela anunciada
- LastByteSent - LastByteAcked ≤ rwnd

7. Encerramento de Conexão - Seção 3.5.6



```
| |
|-------- ACK ------------>|
| |
| CONEXÃO ENCERRADA |
```

3.3.2 Especificação Detalhada do Código

Classe TCPSocket:

```
class SimpleTCPSocket:
 def __init__(self, port):
 """ Inicializa socket UDP subjacente e estruturas de dados """
 self.udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
 self.port = port

 # Estados da conexão
 self.state = 'CLOSED' # CLOSED, LISTEN, SYN_SENT, ESTABLISHED, etc.

 # Números de sequência e ACK
 self.seq_num = random.randint(0, 1000) # ISN (Initial Sequence Number)
 self.ack_num = 0


 # Buffers
 self.send_buffer = []
 self.recv_buffer = []

 # Controle de fluxo
 self.recv_window = 4096 # bytes

 # Controle de tempo
 self.estimated_rtt = 1.0
 self.dev_rtt = 0.5

 # Dados do peer
 self.peer_address = None

 def connect(self, dest_address):
 """ Inicia conexão com three-way handshake """
 pass # A implementar

 def listen(self):
 """ Coloca socket em modo de escuta """
 pass # A implementar

 def accept(self):
 """ Aceita conexão entrante (completa handshake) """
 pass # A implementar

 def send(self, data):
 """ Envia dados (pode bloquear se buffer cheio) """
 pass # A implementar
```

```
 def recv(self, buffer_size):
 """ Recebe dados do buffer de recepção """
 pass # A implementar

 def close(self):
 """ Fecha conexão (four-way handshake) """
 pass # A implementar


 # Métodos auxiliares internos
 def _send_segment(self, flags, data=b''):
 """Cria e envia segmento TCP"""
 pass

 def _receive_loop(self):
 """Thread que recebe segmentos UDP e processa"""
 pass

 def _calculate_timeout(self):
 """Calcula timeout baseado em RTT"""
 return self.estimated_rtt + 4 * self.dev_rtt

 def _update_rtt(self, sample_rtt):
 """Atualiza estimativa de 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)
```

3.3.3 Testes Obrigatórios para Fase 3

01

Teste 1: Estabelecimento de Conexão

```
# Servidor
server = SimpleTCPSocket(8000)
server.listen()
conn = server.accept()


# Cliente
client = SimpleTCPSocket(9000)
client.connect(('localhost', 8000))
```

Verificar captura de pacotes (Wireshark) mostrando SYN, SYN-ACK, ACK

02

Teste 2: Transferência de Dados

```
# Enviar 10KB de dados
data = b'x' * 10240
client.send(data)
received = server.recv(10240)

assert data == received
```

03

Teste 3: Controle de Fluxo

- Reduzir janela do receptor para 1KB
- Enviar 10KB
- Verificar que remetente respeita janela

04

Teste 4: Retransmissão

- Simular perda de 20% dos segmentos
- Verificar que dados são retransmitidos
- Medir tempo total de transferência

05

Teste 5: Encerramento Correto

```
client.close()
server.close()
```

Verificar FIN e ACKs finais

06

Teste 6: Desempenho

- Transferir arquivo de 1MB
- Medir:
  - Throughput (Mbps)
  - Número de retransmissões
  - RTT médio
- Comparar com TCP real (socket padrão Python)

4. ESPECIFICAÇÕES TÉCNICAS

4.1 Linguagem e Bibliotecas

Linguagem obrigatória: Python 3.8+

Bibliotecas permitidas:

- socket (para UDP)
- threading ou asyncio (para concorrência)
- struct (para empacotamento de bytes)
- time (para timers)
- hashlib (para checksums)
- random (para simulação de erros)

Bibliotecas proibidas:

- Qualquer biblioteca que implemente TCP ou protocolos confiáveis prontos

4.2 Estrutura do Projeto

```
projeto_redes/
│
├── fase1/
│ ├── rdt20.py # Implementação rdt2.0
│ ├── rdt21.py # Implementação rdt2.1
│ └── rdt30.py # Implementação rdt3.0
│
├── fase2/
│ ├── gbn.py # Go-Back-N OU
│ └── sr.py # Selective Repeat
│
├── fase3/
│ ├── tcp_socket.py # Classe SimpleTCPSocket
│ ├── tcp_server.py # Aplicação servidor de exemplo
│ └── tcp_client.py # Aplicação cliente de exemplo
│
├── testes/
│ ├── test_fase1.py
│ ├── test_fase2.py
│ └── test_fase3.py
│
├── utils/
│ ├── packet.py # Estruturas de pacotes
│ ├── simulator.py # Simulador de erros e perdas
│ └── logger.py # Sistema de logging
│
├── relatorio/
│ └── relatorio.pdf # Relatório final
│
└── README.md # Instruções de execução
```

4.3 Simulador de Canal Não Confiável

Implemente um simulador que possa:

```
class UnreliableChannel:
 def __init__(self, loss_rate=0.1, corrupt_rate=0.1, delay_range=(0.01,
0.5)):
 """
 loss_rate: probabilidade de perda de pacote (0.0 a 1.0)
 corrupt_rate: probabilidade de corrupção (0.0 a 1.0)
 delay_range: tupla (min_delay, max_delay) em segundos
 """
 self.loss_rate = loss_rate
 self.corrupt_rate = corrupt_rate
 self.delay_range = delay_range

 def send(self, packet, dest_socket, dest_addr):
 """ Envia pacote através do canal não confiável """
 # Simular perda
 if random.random() < self.loss_rate:
 print(f"[SIMULADOR] Pacote perdido")
 return

 # Simular corrupção
 if random.random() < self.corrupt_rate:
 packet = self._corrupt_packet(packet)
 print(f"[SIMULADOR] Pacote corrompido")

 # Simular atraso
 delay = random.uniform(*self.delay_range)
 threading.Timer(delay, lambda: dest_socket.sendto(packet,
dest_addr)).start()

 def _corrupt_packet(self, packet):
 """Corrompe bits aleatórios do pacote"""

 packet_list = list(packet)
 num_corruptions = random.randint(1, 5)
 for _ in range(num_corruptions):
 idx = random.randint(0, len(packet_list) - 1)
 packet_list[idx] = packet_list[idx] ^ 0xFF # Inverter todos os bits
 return bytes(packet_list)
```

5. ENTREGÁVEIS

5.1 Código Fonte

- Código Python bem comentado

- Seguir PEP 8 (padrão de estilo Python)
- Incluir docstrings em todas as funções

5.2 Testes

Cada fase deve incluir:

- Scripts de teste automatizados
- Logs de execução dos testes
- Screenshots ou saída de terminal mostrando testes passando

5.3 Relatório (8-12 páginas)

Estrutura obrigatória:

1. **Introdução**
   - Objetivos da atividade
   - Conceitos teóricos relevantes do Capítulo 3
2. **Fase 1: Protocolos RDT**
   - Descrição da implementação de cada protocolo (rdt2.0, 2.1, 3.0)
   - Diagramas de estados (FSMs)
   - Resultados dos testes
   - Análise: como cada protocolo melhora o anterior
3. **Fase 2: Pipelining**
   - Justificativa da escolha (GBN ou SR)
   - Descrição da implementação
   - Análise de desempenho: gráficos de throughput vs. tamanho da janela
   - Comparação com stop-and-wait
4. **Fase 3: TCP Simplificado**
   - Arquitetura da solução
   - Descrição dos componentes principais
   - Máquina de estados da conexão
   - Resultados dos testes
   - Comparação de desempenho com TCP real
5. **Discussão**
   - Desafios encontrados e soluções
   - Limitações da implementação
   - Diferenças entre TCP simplificado e TCP real
6. **Conclusão**
   - Lições aprendidas
   - Conceitos do Capítulo 3 aplicados na prática
7. **Referências**
   - KUROSE, J. F.; ROSS, K. W. Computer Networking: A Top-Down Approach. 8th ed. Pearson, 2021. Chapter 3.
   - RFCs relevantes (RFC 793 para TCP)

6. CRITÉRIOS DE AVALIAÇÃO

Critério	Pontuação	Descrição
Fase 1: Protocolos RDT	**30 pontos**	
- rdt2.0 funcionando	8	Implementação correta com ACK/NAK
- rdt2.1 funcionando	10	Números de sequência implementados
- rdt3.0 funcionando	12	Timer e perda de pacotes tratados
Fase 2: Pipelining	**25 pontos**	
- Implementação correta	15	GBN ou SR funcionando corretamente
- Testes de eficiência	5	Comparação de desempenho documentada
- Análise de resultados	5	Gráficos e interpretação
Fase 3: TCP sobre UDP	**30 pontos**	
- Three-way handshake	5	Estabelecimento de conexão
- Transferência confiável	10	Dados enviados/recebidos corretamente
- Controle de fluxo	5	Janela respeitada
- Retransmissão adaptativa	5	RTT e timeout implementados
- Encerramento de conexão	5	Four-way handshake
Relatório	**10 pontos**	
- Clareza e organização	4	Estrutura lógica, escrita clara
- Análise técnica	4	Discussão profunda dos resultados
- Referências e formato	2	Citações corretas, formatação
Qualidade do Código	**5 pontos**	
- Organização e estilo	2	comentários (docstrings), estrutura
- Robustez	2	Tratamento de erros
- Documentação	1	README, docstrings
TOTAL	**100 pontos**	

7. DICAS E BOAS PRÁTICAS

7.1 Desenvolvimento Incremental

1. **Não pule etapas**: Implemente rdt2.0 antes de rdt2.1
2. **Teste exaustivamente cada fase** antes de avançar
3. **Mantenha versões funcionais**: use Git para controle de versão

7.2 Debugging

1. **Logging detalhado**: registre cada pacote enviado/recebido
2. **Use Wireshark**: capture tráfego UDP para visualizar pacotes
3. **Simule localmente**: teste em localhost antes de rede real

7.3 Testes

```
# Exemplo de teste automatizado
def test_rdt30_with_loss():
    simulator = UnreliableChannel(loss_rate=0.2)
    sender = RDT30Sender(simulator)
    receiver = RDT30Receiver()

    test_data = [f"Mensagem {i}" for i in range(20)]
    for msg in test_data:
        sender.send(msg)

    # Aguardar conclusão
    time.sleep(10)

    received_data = receiver.get_all_messages()
    assert received_data == test_data
    print("✓ Teste passou: todas as mensagens recebidas corretamente")
```

7.4 Referências do Capítulo 3

Ao implementar, consulte:

- **Figuras 3.9 a 3.16**: FSMs dos protocolos rdt
- **Figuras 3.19, 3.20**: Go-Back-N
- **Figuras 3.23, 3.24**: Selective Repeat
- **Figura 3.28**: Buffers de envio/recepção TCP
- **Figura 3.29**: Estrutura do segmento TCP
- **Figura 3.30**: Números de sequência TCP
- **Seção 3.5.3**: Equações para estimativa de RTT

8. PRAZO E FORMATO DE ENTREGA

Prazo: 09/11/2025 (Não prorrogável)

Formato de Entrega:

1. Arquivo ZIP contendo:
    - Código fonte completo
    - Relatório em PDF
    - README com instruções de execução
    - Logs de testes
2. Nome do arquivo: RDT_TCP_[Nome1]_[Nome2].zip
3. Enviar via Canvas

9. RECURSOS ADICIONAIS

9.1 Ferramentas Recomendadas

- **Wireshark**: Para captura e análise de pacotes UDP
- **IDEs** com bom suporte para Python
- **matplotlib**: Para gráficos de desempenho no relatório
- **pytest**: Para testes automatizados

9.2 Documentação de Referência

- Python Socket Programming: **https://docs.python.org/3/library/socket.html**
- Python Threading: **https://docs.python.org/3/library/threading.html**
- RFC 793 (TCP): **https://www.rfc-editor.org/rfc/rfc793**

10. PERGUNTAS FREQUENTES

P: Posso usar bibliotecas externas?

R: Apenas as listadas na seção 4.1. Para outras, consulte o professor.

P: Preciso implementar todas as funcionalidades do TCP?

R: Não. Apenas as especificadas na Fase 3. Congestion control não é obrigatório.

P: Posso fazer em grupo?

R: grupos de 4-6 alunos

P: O que fazer se meus testes não passarem?

R: Analise os logs, use Wireshark para ver os pacotes, consulte as figuras do livro e, se necessário, tire dúvidas nas aulas.

P: É obrigatório usar Python?

R: Sim, para padronização e facilitar a correção. Se tiver motivo forte para outra linguagem, consulte o professor.

P: Posso implementar GBN E SR?

R: Sim, ganhará pontos extras na avaliação! (até 10 pontos adicionais)

CONCLUSÃO

Esta atividade permitirá que você experimente na prática os conceitos fundamentais de transferência confiável de dados e do protocolo TCP estudados no Capítulo 3 do Kurose. Ao completar as três fases, você terá construído uma implementação funcional (embora simplificada) de um dos protocolos mais importantes da Internet.

> **Lembre-se**: O objetivo não é apenas fazer o código funcionar, mas **entender** os princípios por trás de cada mecanismo. Reflita sobre cada decisão de design e documente suas análises no relatório.

Bom trabalho! 🚀

Elaborado com base em: KUROSE, J. F.; ROSS, K. W. *Computer Networking: A Top-Down Approach*. 8th Edition. Pearson, 2021. Chapter 3: Transport Layer.