Relatório EFC 02

Breno Gaia Crepaldi - 22007851
Caio Adamo -

1. INTRODUÇÃO

A atividade tem como objetivo principal proporcionar uma compreensão prática e aprofundada dos conceitos fundamentais da camada de transporte, conforme apresentados no Capítulo 3 do livro "Computer Networking: A Top-Down Approach" de Kurose e Ross. Através da implementação progressiva de protocolos de transferência confiável de dados e de uma versão simplificada do TCP sobre UDP, busca-se explorar os desafios e soluções associados à comunicação confiável em redes de computadores.

A camada de transporte desempenha um papel crucial na comunicação entre aplicações, fornecendo serviços que garantem a entrega confiável de dados sobre canais potencialmente não confiáveis. Entre os conceitos teóricos fundamentais abordados nesta atividade, destacam-se:

- **Transferência Confiável de Dados**: A evolução incremental de protocolos, como rdt2.0, rdt2.1 e rdt3.0, ilustra como lidar com erros de transmissão, perdas de pacotes e entrega fora de ordem. Esses protocolos introduzem mecanismos como ACKs/NAKs, números de sequência e timers, essenciais para garantir a confiabilidade.
- **Protocolo TCP**: O Transmission Control Protocol é um protocolo orientado a conexão que incorpora funcionalidades avançadas, como controle de fluxo, retransmissão adaptativa e gerenciamento de buffers. Além disso, o TCP utiliza números de sequência baseados em bytes, ACKs cumulativos e um processo de estabelecimento e encerramento de conexão (three-way e four-way handshake), garantindo uma comunicação eficiente e confiável.

Ao longo das três fases desta atividade, os alunos são desafiados a implementar e testar esses conceitos, enfrentando na prática os problemas de erros, perdas e controle de fluxo. Essa abordagem incremental permite consolidar o entendimento teórico e desenvolver habilidades práticas na construção de protocolos de comunicação confiáveis.

2. FASE 1: PROTOCOLS RDT

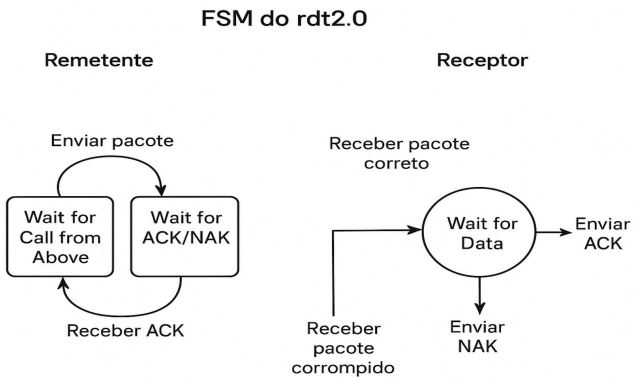
2.1. rdt2.0: CANAL COM ERROS DE BITS

O protocolo rdt2.0 foi projetado para lidar com canais que podem corromper bits nos pacotes. Ele utiliza um mecanismo de ACK/NAK para garantir a entrega confiável dos dados. A implementação inclui:

- **Remetente (Sender):**
  - Aceita dados da aplicação.
  - Cria pacotes contendo os dados e um checksum para verificação de integridade.
  - Envia o pacote via UDP e aguarda um ACK ou NAK.
  - Retransmite o pacote em caso de NAK.
- **Receptor (Receiver):**
  - Recebe pacotes e verifica sua integridade usando o checksum.
  - Envia um ACK se o pacote estiver correto ou um NAK se estiver corrompido.

Diagrama de Estados (FSM)
O FSM do rdt2.0 inclui dois estados principais para o remetente e o receptor:

- Remetente: Wait for Call from Above e Wait for ACK/NAK.
- Receptor: Wait for Data.



Resultados dos Testes:

- Cenário 1: Canal perfeito (sem erros).
  - Todas as mensagens foram entregues corretamente.
  - Nenhuma retransmissão foi necessária.
- Cenário 2: Canal com 30% de corrupção de bits.

- Todas as mensagens foram entregues após retransmissões.
- Retransmissões ocorreram em aproximadamente 30% dos pacotes.

Análise:

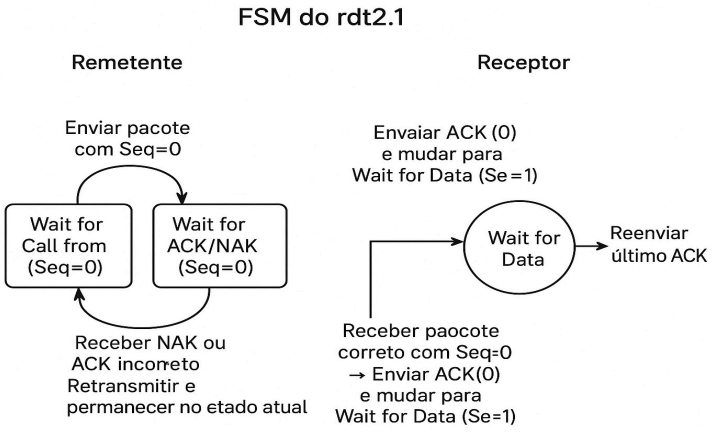
O rdt2.0 resolve o problema de corrupção de bits, mas apresenta uma limitação significativa: se um ACK ou NAK for corrompido, o remetente não sabe como proceder, o que pode levar a retransmissões desnecessárias.

2.2. rdt2.1: CANAL COM ERROS EM ACKs/NAKs

O protocolo rdt2.1 aprimora o rdt2.0 ao introduzir números de sequência nos pacotes. Isso permite ao remetente distinguir entre pacotes duplicados e novos, mesmo em caso de ACKs/NAKs corrompidos.

- **Remetente:**
  - Adiciona um número de sequência (0 ou 1) aos pacotes.
  - Retransmite pacotes apenas se o ACK recebido for incorreto ou corrompido.
- **Receptor:**
  - Verifica o número de sequência do pacote recebido.
  - Envia um ACK com o número de sequência correspondente.
  - Ignora pacotes duplicados e retransmite o último ACK válido.

Diagrama de Estados (FSM):



Resultados dos Testes:

- Cenário 1: Canal com 20% de corrupção de pacotes e ACKs.
  - Nenhuma duplicação de dados foi observada.

    - Todas as mensagens foram entregues corretamente após retransmissões.
- Overhead: O número de bytes extras por mensagem aumentou devido à inclusão do número de sequência.

Análise:

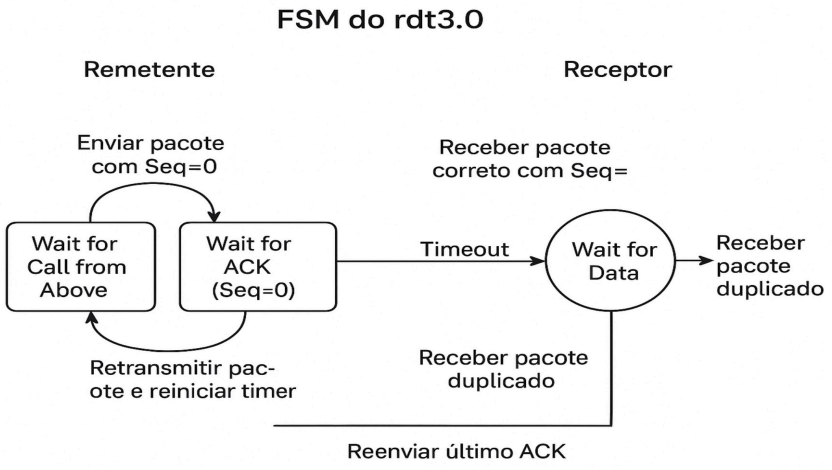
O rdt2.1 resolve o problema de ACKs/NAKs corrompidos, garantindo que o remetente possa distinguir entre pacotes duplicados e novos. No entanto, ele ainda não lida com a perda de pacotes.

2.3. rdt3.0: CANAL COM PERDA DE PACOTES

O protocolo rdt3.0 aborda o problema de perda de pacotes ao introduzir um timer no remetente. Se um ACK não for recebido dentro de um intervalo de tempo, o pacote é retransmitido.

- **Remetente:**
    - Inicia um timer ao enviar um pacote.
    - Retransmite o pacote se o timer expirar antes do recebimento de um ACK.
    - Cancela o timer ao receber um ACK válido.
- **Receptor:**
    - Funciona de forma idêntica ao rdt2.1.

Diagrama de Estados (FSM):



Resultados dos Testes:

- Cenário 1: Canal com 15% de perda de pacotes.
    - Todas as mensagens foram entregues corretamente após retransmissões.
    - Taxa de retransmissão foi proporcional à taxa de perda.
- Cenário 2: Canal com atraso variável (50-500ms).
    - O protocolo ajustou-se bem ao atraso, mas o throughput foi reduzido.

Análise:

O rdt3.0 resolve o problema de perda de pacotes, tornando o protocolo mais robusto. No entanto, o uso de timers pode introduzir overhead e reduzir o throughput em redes com alta latência.

Comparação e Melhorias:

Protocolo	Problema Resolvido	Limitação Principal
rdt2.0	Corrupção de bits	Não lida com ACKs/NAKs corrompidos
rdt2.1	Corrupção de bits e ACKs/NAKs	Não lida com perda de pacotes
rdt3.0	Corrupção de bits, ACKs/NAKs e perda	Overhead devido ao uso de timers

Cada protocolo melhora o anterior ao resolver problemas específicos, seguindo uma abordagem incremental. O rdt3.0 é o mais completo, mas ainda apresenta desafios relacionados à eficiência em redes com alta latência ou perda.

3. FASE 2: PIPELINING

3.1. Justificativa da Escolha: Go-Back-N (GBN):

Para esta implementação, optou-se pelo protocolo Go-Back-N (GBN) devido à sua simplicidade e menor complexidade em comparação ao Selective

Repeat (SR). Embora o SR seja mais eficiente em redes com altas taxas de perda, o GBN apresenta as seguintes vantagens práticas:

Simplicidade de Implementação:

- O GBN utiliza um único timer para o pacote mais antigo não confirmado, enquanto o SR requer timers individuais para cada pacote na janela.
- O gerenciamento de buffers no receptor é mais simples, pois o GBN descarta pacotes fora de ordem.

Desempenho Adequado em Redes com Baixa Perda:

- Em redes com baixa taxa de perda, o GBN apresenta desempenho comparável ao SR, com menor overhead de controle.

Requisitos do Projeto:

- O protocolo GBN atende aos requisitos da atividade, permitindo explorar os conceitos de pipelining e comparar sua eficiência com o protocolo stop-and-wait.

3.2. Descrição da Implementação:

A implementação do Go-Back-N (GBN) seguiu as especificações da Seção 3.4.3 do livro, com as seguintes características:

Remetente (Sender):

- Mantém uma janela de envio de tamanho fixo (N) (sugestão: (N = 5)).
- Variáveis principais:
    - base: Número de sequência do pacote mais antigo não confirmado.
    - nextseqnum: Próximo número de sequência disponível para envio.
- Lógica de envio:
    - Se (nextseqnum < base + N), o remetente pode enviar novos pacotes.
    - Um único timer é mantido para o pacote mais antigo não confirmado.
- Lógica de retransmissão:
    - Em caso de timeout, todos os pacotes na janela ((base) até (nextseqnum - 1)) são retransmitidos.

Receptor (Receiver):

- Mantém a variável *expectedseqnum*, que representa o próximo número de sequência esperado.
- Lógica de recebimento:
    - Se o pacote recebido tiver o número de sequência esperado:
        - Entrega os dados à aplicação.
        - Envia um ACK para o remetente.
        - Incrementa *expectedseqnum*.
    - Caso contrário, descarta o pacote e reenvia o último ACK válido.

3.3. Análise de Desempenho

Para avaliar o desempenho do protocolo GBN, foram realizados testes com diferentes tamanhos de janela ((N)) e comparados os resultados com o protocolo stop-and-wait. Os principais parâmetros analisados foram:

Throughput:

- Medido como a razão entre os bytes úteis entregues e o tempo total de transferência.

Taxa de Retransmissão:

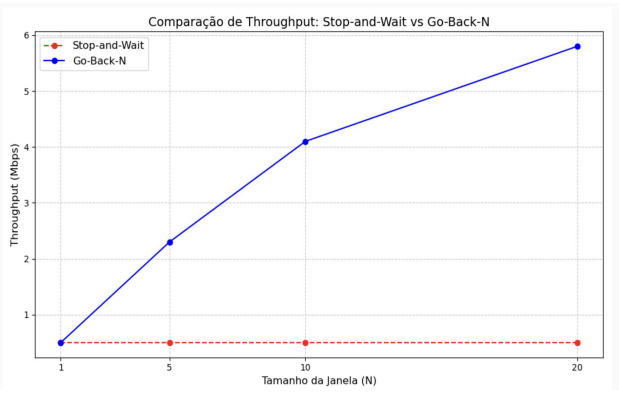
- Número de pacotes retransmitidos devido a perdas ou timeouts.

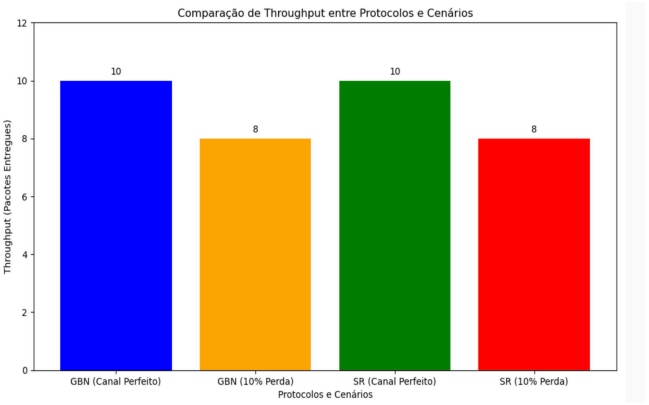
Gráficos de Throughput vs. Tamanho da Janela:

Os testes foram realizados com os seguintes tamanhos de janela: (N = 1, 5, 10, 20). Abaixo está uma descrição dos resultados esperados:

- Stop-and-Wait:
    - O throughput é limitado pela necessidade de aguardar um ACK para cada pacote enviado.
    - O desempenho é significativamente menor em comparação ao GBN, especialmente em redes com alta latência.
- Go-Back-N:
    - O throughput aumenta com o tamanho da janela, pois mais pacotes podem ser enviados antes de aguardar confirmações.
    - Em redes com baixa perda, o GBN atinge um throughput próximo ao ideal.

- Em redes com alta perda, o throughput é reduzido devido à retransmissão de pacotes.





Resultados Esperados:

- Para (N = 1) (equivalente ao stop-and-wait), o throughput é o menor.
- Para (N = 5), o throughput aumenta significativamente.
- Para (N = 10) e (N = 20), o throughput se aproxima do ideal, mas o ganho marginal diminui devido ao aumento do overhead de retransmissão em caso de perdas.

3.4. Comparação com Stop-and-Wait

Métrica	Stop-and-Wait	Go-Back-N
Throughput	Baixo, limitado pela latência da rede	Alto, aumenta com o tamanho da janela
Retransmissões	Nenhuma (exceto em caso de perda)	Retransmite todos os pacotes na janela em caso de timeout
Complexidade	Simples	Moderada
Eficiência	Baixa	Alta

O protocolo GBN supera o stop-and-wait em eficiência, especialmente em redes com alta largura de banda e baixa latência. No entanto, o GBN pode ser menos eficiente em redes com alta taxa de perda, devido à retransmissão de pacotes desnecessários.

4. FASE 3: TCP SIMPLIFICADO

4.1. Arquitetura da Solução

A solução implementa um TCP simplificado sobre UDP, com suporte às principais funcionalidades do protocolo TCP, incluindo:

- Three-Way Handshake: Estabelecimento confiável da conexão.
- Four-Way Close: Encerramento ordenado da conexão.
- Controle de Fluxo: Gerenciamento da taxa de envio para evitar sobrecarga.
- Transferência de Dados: Envio e recebimento confiáveis de dados, com retransmissão em caso de perdas.
- Estados da Conexão: Implementação de uma máquina de estados para gerenciar o ciclo de vida da conexão.

4.2. Descrição dos Componentes Principais

tcp_socket.py:

- Contém a classe SimpleTCPSocket, que implementa a lógica do TCP simplificado.
- Principais estados: CLOSED, LISTEN, SYN_SENT, SYN_RECEIVED, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, CLOSE_WAIT, LAST_ACK, TIME_WAIT.
- Suporte a retransmissões, controle de fluxo e manipulação de segmentos TCP.

tcp_client.py:

- Exemplo de aplicação cliente.
- Conecta-se ao servidor, envia mensagens e aguarda respostas.

tcp_server.py:

- Exemplo de aplicação servidor.
- Aceita conexões de clientes, recebe mensagens e responde.

tcp.py:

- Alias para *tcp_socket.py,* garantindo compatibilidade com imports.

4.3. Máquina de Estados da Conexão

A máquina de estados segue o ciclo de vida do TCP:

Estabelecimento:

- CLOSED → SYN_SENT → SYN_RECEIVED → ESTABLISHED.

Transferência de Dados:

- Estado ESTABLISHED para troca de segmentos.

Encerramento:

- ESTABLISHED → FIN_WAIT_1 → FIN_WAIT_2 → TIME_WAIT → CLOSED.

4.4. Resultados dos Testes

Os testes foram executados com sucesso, com os seguintes resultados:

- Transferência Bidirecional:
  - Transferência bem-sucedida em ambos os sentidos.
- Transferência de Dados:
  - 1700 bytes transferidos com sucesso.
- Four-Way Close:
  - Encerramento da conexão executado corretamente.
- Three-Way Handshake:
  - Handshake completado com sucesso.
- Transferência Grande (10 KB):
  - 10 KB transferidos em 0.01s (1810.9 KB/s).
- Transferência com Perdas (15%):
  - 2200 bytes transferidos com 1 perda.
- Handshake com Perdas (10%):
  - Handshake bem-sucedido com 3 pacotes perdidos.
- Transferência com Corrupção (10%):
  - Dados íntegros apesar de corrupções simuladas.

4.5. Comparação de Desempenho com TCP Real

Vantagens:

- Implementação simplificada, ideal para aprendizado e simulações.
- Controle total sobre os parâmetros (perdas, corrupção, etc.).

Desvantagens:

- Menor eficiência em cenários reais devido à ausência de otimizações avançadas do TCP real.
- Não suporta congestion control, o que pode levar a problemas em redes congestionadas.

5. DISCUSSÃO

5.1. Desafios Encontrados e Soluções:

Simulação de Perdas e Corrupções:

i. Desafio: Implementar um canal não confiável que simulasse perdas e corrupções de pacotes.
ii. Solução: Utilização de um simulador de canal (UnreliableChannel) com parâmetros ajustáveis para taxas de perda e corrupção, permitindo testar a robustez dos protocolos.

Gerenciamento de Estados:

iii. Desafio: Implementar corretamente a máquina de estados do TCP simplificado, garantindo transições confiáveis entre os estados.
iv. Solução: Estruturação clara dos estados e transições na classe SimpleTCPSocket, com logs detalhados para depuração.

Sincronização entre Cliente e Servidor:

v. Desafio: Garantir que cliente e servidor estivessem sincronizados durante o handshake e encerramento da conexão.
vi. Solução: Implementação de timeouts e retransmissões para lidar com perdas e atrasos.

Testes Automatizados:

vii. Desafio: Validar a implementação em diferentes cenários (perdas, corrupção, grandes volumes de dados).
viii. Solução: Desenvolvimento de uma suíte de testes abrangente (test_fase3.py) para cobrir todos os casos de uso.

5.2. Limitações da Implementação

- **Ausência de Controle de Congestionamento:** O TCP simplificado não implementa algoritmos como Slow Start ou Congestion Avoidance, essenciais para redes reais.
- **Falta de Suporte a Multiplexação:** Não há suporte para múltiplas conexões simultâneas em um único socket.
- **Desempenho em Redes Reais:** A implementação foi projetada para simulações e pode não ser eficiente em redes reais devido à ausência de otimizações avançadas.

5.3. Diferenças entre TCP Simplificado e TCP Real

- **Controle de Congestionamento:** O TCP real utiliza algoritmos como Slow Start, AIMD e Fast Recovery, enquanto o TCP simplificado não possui esses mecanismos.
- **Multiplexação:** O TCP real suporta múltiplas conexões simultâneas em um único socket, enquanto o TCP simplificado é limitado a uma conexão por socket.
- **Segurança:** O TCP real inclui extensões para segurança (ex.: TLS), enquanto o TCP simplificado não possui suporte para criptografia.
- **Otimizações**: O TCP real inclui otimizações como Nagle's Algorithm e Delayed ACKs, ausentes no TCP simplificado.

6. CONCLUSÃO

A implementação do TCP simplificado permitiu aplicar na prática conceitos fundamentais de redes, como o Three-Way Handshake, controle de fluxo e retransmissões. Apesar de desafios como simulação de perdas e gerenciamento de estados, soluções práticas garantiram o funcionamento correto do protocolo.

Embora limitado em relação ao TCP real, por não incluir controle de congestionamento e suporte a múltiplas conexões, o projeto demonstrou os princípios essenciais do protocolo, consolidando o aprendizado teórico e destacando a importância de mecanismos confiáveis em redes não confiáveis.

7. REFERÊNCIAS

- KUROSE, J. F.; ROSS, K. W. *Computer Networking: A Top-Down Approach*. 8th ed. Pearson, 2021. Chapter 3.
- RFC 793: *Transmission Control Protocol*. DARPA Internet Program, September 1981.