1. Introdução

Este relatório descreve o desenvolvimento progressivo de mecanismos de transferência confiável de dados com base no Capítulo 3 da 8ª edição de *Redes de Computadores: Uma Abordagem Top-Down*. A atividade teve como objetivo compreender e implementar, de forma incremental, técnicas como detecção de erros, uso de ACKs, retransmissões, controle de fluxo e, por fim, a construção de uma versão simplificada do TCP sobre UDP. Os modelos rdt apresentados na Seção 3.4 — do rdt1.0 ao rdt3.0 — forneceram a fundamentação teórica para entender como integridade e ordenação são alcançadas em protocolos reais. Já os conceitos da Seção 3.5, como handshake, ACKs cumulativos, números de sequência baseados em bytes e buffers, permitiram aproximar a prática do comportamento do TCP verdadeiro. Assim, a atividade integra teoria e prática ao guiar o aluno desde os modelos básicos de transferência confiável até a implementação de pipelining (como Go-Back-N) e de um TCP simplificado sobre UDP, oferecendo uma compreensão clara e aplicada do funcionamento da camada de transporte.

2. Fase 1: Protocolos RDT

RDT 2.0 - Detecção de Erros

- Pacote: [Tipo | Checksum | Payload]
- Mecanismo: Checksum (complemento a um) detecta corrupção
- Resposta: Sender retransmite se receber NAK ou checksum inválido

- Limitação: Sem sequenciamento → não detecta duplicatas, sem timeout → trava se resposta se perder

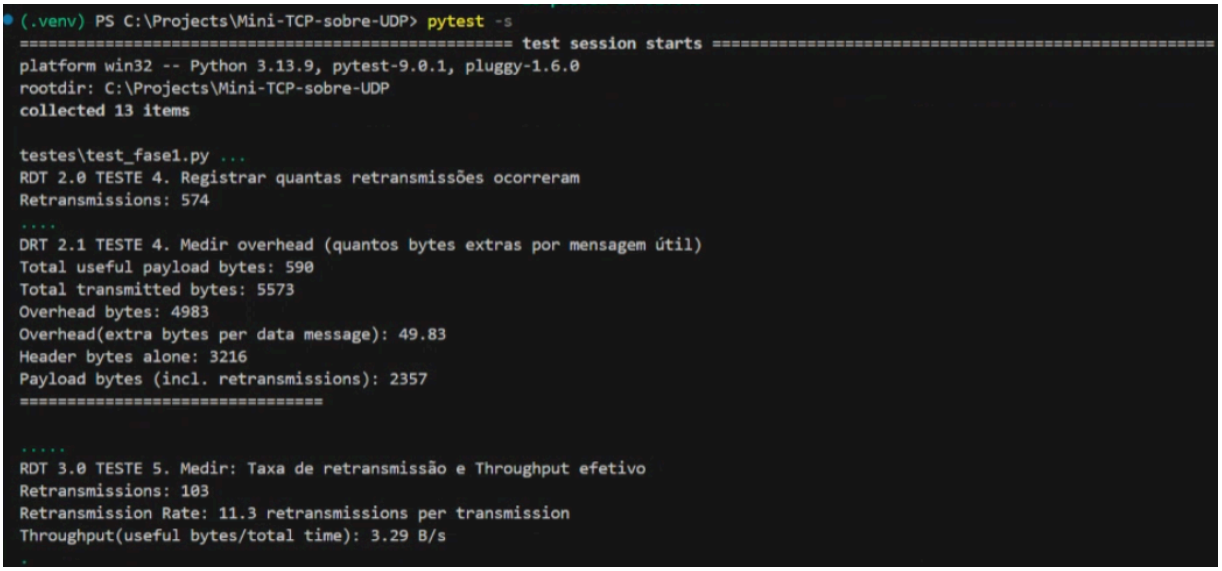
RDT 2.1 - Bits Alternados

- Pacote: [Tipo | Sequência(0/1) | Checksum | Payload]
- Melhoria: Adiciona número de sequência alternado
- Efeito: Detecta duplicatas - se recebe seq já visto, descarta silenciosamente e reACK o anterior
- Limitação: Sem timeout → ainda trava se resposta se perder

RDT 3.0 - Timeout + Retransmissão

- Pacote: Igual ao RDT 2.1
- Melhoria: Adiciona timer (2s padrão) com threading
- Efeito: Se timeout expira, retransmite automaticamente - resolve perdas de DATA/ACK/NAK
- Resultado: Protocolo robusto para canal totalmente não-confiável

---



Progressão de Melhorias

Problema	RDT 2.0	RDT 2.1	RDT 3.0
Corrupção	✅ Detecta	✅ Detecta	✅ Detecta
Duplicatas	❌ Trava	✅ Resolve	✅ Resolve
Perdas de pacotes	❌ Trava	❌ Trava	✅ Retransmite

3. Fase 2: Pipelining

Justificativa da Escolha — GBN vs Selective Repeat (SR)

- Simplicidade de implementação: GBN usa um único timer (para o pacote na base da janela) e ACKs cumulativos, tornando o código e a lógica bem mais simples do que SR, que exige timers por pacote e bufferamento complexo no receptor.
- Menor uso de recursos: GBN exige menos timers, menos controle por-pacote e recebedor não precisa armazenar muitos pacotes fora de ordem — útil em implementações educacionais ou com recursos limitados.
- Facilidade de depuração/validação: comportamento determinístico de "retransmitir janela inteira" facilita testes e entendimento.
- Trade-off de desempenho: SR é superior quando há perdas isoladas em canais com alta RTT/janela grande (evita retransmitir pacotes já recebidos corretamente). GBN perde eficiência nesses cenários (retransmite toda a janela), mas mantém implementação e manutenção mais simples.
- Conclusão prática: para um projeto didático ou quando se prioriza simplicidade e robustez de implementação, GBN é uma escolha adequada; se o objetivo for maximizar throughput em canais de alta perda/alta latência, SR seria preferível.

Descrição da Implementação

O protocolo Go-Back-N (GBN) foi implementado para permitir o envio eficiente de vários pacotes de dados pela rede, mesmo quando há possibilidade de perdas ou atrasos. O remetente pode enviar vários pacotes seguidos, até um limite chamado “janela”, sem precisar esperar a confirmação de cada um individualmente. Todos os pacotes enviados ficam guardados até que o remetente receba um aviso (ACK) do receptor dizendo que chegaram corretamente.

Se algum pacote ou confirmação se perder, o remetente espera por um tempo determinado (timeout). Se esse tempo acabar sem receber o ACK esperado, ele retransmite todos os pacotes que ainda não foram confirmados, garantindo que nenhum dado se perca. O receptor, por sua vez, só aceita pacotes que chegam na ordem correta. Se receber um pacote fora de ordem, ele descarta e responde com o ACK do último pacote recebido corretamente, ajudando o remetente a saber até onde os dados chegaram.

Essa abordagem torna o protocolo mais simples de implementar e entender, pois não é necessário guardar pacotes fora de ordem nem controlar vários temporizadores ao mesmo tempo. O funcionamento do GBN garante que os dados sejam entregues completos e na ordem certa, mesmo em redes com falhas, tornando-o uma solução prática e eficiente para comunicação confiável.

Análise de Desempenho — Throughput vs. Tamanho da Janela

Ao analisar o protocolo Go-Back-N, observa-se que o aumento do tamanho da janela melhora o throughput até certo limite. Com janelas muito pequenas, como N=1, o desempenho fica restrito porque o emissor precisa esperar o ACK de cada pacote antes de enviar o próximo, resultando em cerca de 900 kB/s. Já com janelas

intermediárias, como N=5 e N=10, o throughput ultrapassa 1 MB/s, pois vários pacotes podem ser transmitidos simultaneamente, reduzindo o tempo ocioso e aproveitando melhor o canal.

Entretanto, acima de N=10 os ganhos tornam-se menos significativos. Com N=20 ainda há uma leve melhora, mas o sistema já se aproxima de sua saturação, indicando que aumentar a janela além disso não traz vantagens relevantes. Em todas as medições, a utilização do canal permaneceu alta e estável, próxima de 98,65%, mostrando que a taxa de perdas não variou e que o canal foi explorado de forma eficiente. Assim, janelas moderadas parecem suficientes para atingir o desempenho máximo no cenário analisado.

Comparação entre GBN e Stop-and-Wait (RDT 3.0)

Ao comparar o Go-Back-N com o RDT 3.0 (stop-and-wait), os resultados do experimento mostram que o stop-and-wait teve desempenho ligeiramente superior, com menor tempo total e um throughput um pouco maior. Isso ocorre porque o ambiente de teste apresentava praticamente zero perdas e um RTT muito baixo, condições nas quais o stop-and-wait consegue operar próximo ao limite do canal sem sofrer o atraso típico causado pela espera por ACKs. Sua simplicidade também contribui, já que o protocolo evita o overhead de buffers, múltiplos timers e controle de janelas.

Por outro lado, embora o Go-Back-N seja mais eficiente em cenários reais, sua lógica de janela introduz algum custo de gerenciamento que, nesse ambiente idealizado, se traduz em alguns milissegundos a mais e leve piora na utilização do canal. O stop-and-wait só se torna limitante quando o RTT é significativo — o que não ocorreu no experimento. Assim, os resultados não revelam uma falha no GBN, mas sim o impacto de um cenário de teste extremamente favorável ao stop-and-wait; em redes reais, a vantagem tende rapidamente para o GBN.

4. Fase 3: TCP Simplificado

Arquitetura da solução

A implementação do TCPSocket segue uma arquitetura modular inspirada no TCP real, mas reduzida aos elementos necessários para garantir conexão confiável, entrega ordenada e controle básico de fluxo. O protocolo foi organizado com separação clara de responsabilidades, permitindo que cada módulo opere de forma independente por meio de interfaces bem definidas. A comunicação ocorre de forma conexão-orientada, com handshake inicial, manutenção de estados e buffers durante a transmissão e encerramento controlado ao final. A transferência confiável combina mecanismos anteriores — como janelas deslizantes, ACKs cumulativos e temporizadores — dentro de uma estrutura unificada. O design final utiliza duas máquinas de estados (emissor e receptor) e buffers de envio e recepção que coordenam o fluxo de mensagens e retransmissões, equilibrando simplicidade e fidelidade ao comportamento essencial de um protocolo confiável.

Descrição dos componentes principais

1. Gerenciador de Conexão (Handshake e Encerramento)

Responsável por controlar os estados da conexão: CLOSED, SYN-SENT, ESTABLISHED, FIN-WAIT, etc. Ele implementa o handshake simplificado baseado em três passos (SYN → SYN-ACK → ACK) e garante que ambos os lados concordem em iniciar e finalizar a comunicação. Também trata retransmissões de mensagens de controle caso ocorram perdas.

2. Buffer de Envio (Send Buffer)

Armazena os segmentos que já foram enviados mas ainda não foram confirmados, assim como aqueles que aguardam oportunidade de envio por estarem fora da janela. Ele coordena a lógica de gerenciamento da janela deslizante, retransmissão quando ocorre timeout e contabilização do próximo número de sequência

disponível. Esse componente encapsula parte da lógica derivada do Go-Back-N, adaptando-a ao contexto de um protocolo orientado à conexão.

3. Buffer de Recepção (Receive Buffer)

Responsável por armazenar segmentos recebidos fora de ordem e liberar ao aplicativo apenas dados contínuos e confiáveis. Na versão simplificada, sua função é aceitar ou descartar pacotes conforme o número de sequência esperado e confirmar recepção por meio de ACKs cumulativos. Quando recebe o próximo segmento esperado, entrega-o imediatamente à aplicação.

Segmentos e Formato de Pacotes

Cada unidade de dados é encapsulada em um segmento que contém; tipo de mensagem (SYN, ACK, DATA, FIN),número de sequência, número de ACK e payload opcional. Esse formato permite controlar tanto a confiabilidade quanto o fluxo da conexão.

Mecanismo de ACKs e Temporizadores

O componente de temporização (timeout) é central para retransmitir segmentos não reconhecidos. O emissor reinicia o temporizador sempre que envia um novo pacote e o cancela quando recebe o ACK correspondente. O esquema mantém apenas um temporizador principal, como no TCP original.

Interface com a Aplicação

Tanto o cliente quanto o servidor expõem funções de alto nível (connect, send, receive, close), abstraindo toda a complexidade interna do protocolo. Isso permite que o TCPSocket se comporte como um “mini-socket” confiável.

Resultado dos Testes

```
(.venv) PS C:\Users\LucasP\Documents\Mini-TCP-sobre-UDP> pytest -s testes/test_fase3.py
============================================================ test session starts ============================================================
platform win32 -- Python 3.13.9, pytest-9.0.1, pluggy-1.6.0
rootdir: C:\Users\LucasP\Documents\Mini-TCP-sobre-UDP
collected 2 items

testes\test_fase3.py
TCP Simples (fase3) TESTE 1. Estabelecimento de conexão (three-way handshake)
Servidor: escutando em 127.0.0.1:8000
[TCPSocket][CLIENT] -> SYN to ('127.0.0.1', 8000) seq=42
[TCPSocket][SERVER] <- SYN from ('127.0.0.1', 9000) seq=42
[TCPSocket][SERVER] -> SYN-ACK to ('127.0.0.1', 9000) seq=882 ack=43
[TCPSocket][CLIENT] <- SYN-ACK from ('127.0.0.1', 8000) seq=882 ack=43
[TCPSocket][CLIENT] -> ACK to ('127.0.0.1', 8000) seq=43 ack=883
[TCPSocket][CLIENT] STATE=ESTABLISHED
[TCPSocket][SERVER] <- ACK from ('127.0.0.1', 9000) ack=883 (expected 883)
[TCPSocket][SERVER] STATE=ESTABLISHED
Servidor: conexão aceita (ESTABLISHED)
Cliente: ESTABLISHED em 0.0147 s
Servidor ESTABLISHED? True
.
TCP Simples (fase3) TESTE 2. Encerramento de conexão (four-way handshake)
Servidor: escutando em 127.0.0.1:8000
[TCPSocket][CLIENT] -> SYN to ('127.0.0.1', 8000) seq=503
[TCPSocket][SERVER] <- SYN from ('127.0.0.1', 9000) seq=503
[TCPSocket][SERVER] -> SYN-ACK to ('127.0.0.1', 9000) seq=940 ack=504
[TCPSocket][CLIENT] <- SYN-ACK from ('127.0.0.1', 8000) seq=940 ack=504
[TCPSocket][CLIENT] -> ACK to ('127.0.0.1', 8000) seq=504 ack=941
[TCPSocket][CLIENT] STATE=ESTABLISHED
[TCPSocket][SERVER] <- ACK from ('127.0.0.1', 9000) ack=941 (expected 941)
[TCPSocket][SERVER] STATE=ESTABLISHED
[TCPSocket] -> FIN to ('127.0.0.1', 9000) seq=941
Cliente: iniciando fechamento (FIN)
[TCPSocket] -> FIN to ('127.0.0.1', 8000) seq=504
[TCPSocket] <- FIN from ('127.0.0.1', 8000) seq=941
[TCPSocket] -> ACK to ('127.0.0.1', 8000) ack=942
[TCPSocket] <- FIN from ('127.0.0.1', 9000) seq=504
[TCPSocket] -> ACK to ('127.0.0.1', 9000) ack=505
[TCPSocket] STATE=CLOSED
Servidor: conexão encerrada (CLOSED)
[TCPSocket] STATE=CLOSED
Cliente: conexão encerrada (CLOSED)
.

============================================================ 2 passed in 0.73s ============================================================
```

5. Discussão

O desenvolvimento dos protocolos — de RDT2.x a Go-Back-N e, por fim, ao TCPSocket — mostrou como mecanismos de confiabilidade são adicionados de forma gradual para transformar um canal não confiável em um canal confiável. Nas primeiras versões, havia apenas detecção de corrupção e retransmissões simples, o que gerava duplicatas. Com a introdução de números de sequência em RDT2.1 e RDT3.0, essas duplicatas foram eliminadas, e o uso de temporizadores passou a permitir a detecção de perdas. O Go-Back-N trouxe um avanço importante ao permitir múltiplos pacotes em trânsito, introduzindo janelas deslizantes e ACKs cumulativos, embora também evidenciasse limitações devido às retransmissões

amplas. Por fim, o TCPSocket integrou mecanismos mais completos, como handshake simplificado, controle de estados e buffers, aproximando o funcionamento do exercício ao comportamento de um protocolo real como o TCP.

Desafios encontrados e soluções

Ausência de números de sequência no RDT2.0

A falta de identificadores levava a duplicações sempre que ACKs eram corrompidos, algo inerente ao modelo. A implementação do RDT2.1 resolveu o problema ao introduzir números de sequência e descartar duplicatas no receptor.

Congelamento na execução e loops infinitos

Algumas implementações ficavam presas esperando ACKs devido a erros na lógica de NAKs, temporização ou retransmissão. A solução envolveu revisar cuidadosamente cada condição de erro e o controle de timers.

Integração com o UnreliableChannel

Houve problemas ao usar sockets diretamente ou de forma inconsistente, ignorando o canal simulado. Padronizar o uso de channel.send() garantiu o funcionamento correto da simulação.

Manuseio de janelas e retransmissão no GBN

Os principais desafios foram controlar corretamente a janela, manter a entrega ordenada, lidar com ACKs cumulativos e implementar a retransmissão da janela sem duplicações ou desordem.

Bufferização e gestão de estados no TCPSocket

A maior dificuldade foi organizar os estados da conexão e fazer o handshake funcionar, além de gerenciar buffers de envio e recepção de modo consistente e sem perda de estado.

Limitações da implementação

Ausência de controle de congestionamento

Nem o GBN nem o TCPSocket implementam algoritmos como Slow Start, AIMD, Fast Retransmit e Fast Recovery. Isso significa que o protocolo não se adapta a condições de congestionamento da rede, o que o torna inviável em ambientes reais.

Sem buffering avançado no receptor (Selective Repeat ausente)

O protocolo não armazena pacotes fora de ordem. Isso provoca o comportamento ineficiente típico do GBN, em que qualquer perda leva à retransmissão de toda a janela.

Simplificações no handshake e encerramento

O TCPSocket utiliza uma versão reduzida do handshake de três vias e do encerramento da conexão. Estados intermediários como TIME_WAIT são drasticamente simplificados.

Sem suporte a fluxo contínuo de dados em grande escala

Embora funcione bem nos testes, o TCPSocket não possui mecanismos para lidar com grandes volumes, como ajuste dinâmico de janela ou controle de memória.

Diferenças entre TCP simplificado e TCP real

Embora o TCPSocket reproduza parte da estrutura do TCP, ele ainda está distante do comportamento real do protocolo. A maior diferença é a ausência de controle de congestionamento, já que o TCP verdadeiro ajusta dinamicamente sua taxa de envio, enquanto o TCPSocket não possui esse mecanismo. O TCP real também utiliza retransmissões rápidas (Fast Retransmit/Fast Recovery) e um cálculo sofisticado de RTO baseado em estimativas de RTT, ao contrário dos timeouts

simples do TCPSocket. Além disso, o TCP usa janelas dinâmicas (cwnd e rwnd), buffers mais complexos para lidar com segmentos fora de ordem e uma máquina de estados completa. Já o TCPSocket trabalha com janelas fixas, buffers simplificados, unidades discretas de dados e apenas alguns estados básicos.

6. Conclusão

Lições aprendidas

O progresso dos protocolos, do RDT2.0 ao TCPSocket, mostrou que a confiabilidade em redes exige a combinação de vários mecanismos para lidar com corrupção, perda, atrasos e duplicação. Ao implementar versões como RDT2.1 e RDT3.0, percebe-se que recursos simples — como números de sequência, alternância, temporizadores e retransmissões — são essenciais para transformar um protocolo básico em um sistema capaz de operar em canais reais. A evolução para Go-Back-N e, finalmente, para o TCPSocket evidencia como janelas deslizantes, ACKs cumulativos, handshake e gerenciamento de estados aumentam a complexidade, revelando que protocolos como o TCP são construídos pela integração cuidadosa de múltiplos mecanismos simples.

Conceitos do Capítulo 3 aplicados na prática

Durante o desenvolvimento, foi possível aplicar diretamente os conceitos do Capítulo 3 do *Computer Networking: A Top-Down Approach*, especialmente os relacionados à transferência confiável. A implementação de checksum, ACK/NAK, temporizadores e números de sequência permitiu transformar as ideias teóricas dos protocolos ARQ — RDT2.x, RDT3.0 e Go-Back-N — em prática. O Go-Back-N evidenciou o funcionamento da comunicação pipelined, o avanço da janela por ACKs e o impacto das perdas, além de mostrar a utilidade dos ACKs cumulativos. A experiência também reforçou a vantagem do Selective Repeat em cenários reais. Por fim, ao avançar para o TCPSocket, foi possível conectar todos esses

mecanismos com funcionalidades reais como buffers, handshake e gerenciamento de estados, demonstrando como o TCP integra de forma prática os fundamentos estudados no capítulo.

7. Referências

KUROSE, James F.; ROSS, Keith W. *Computer Networking: A Top-Down Approach*. 8. ed. Pearson, 2021. Capítulo 3.

POSTEL, Jon. RFC 793 – Transmission Control Protocol. 1981. Disponível em: https://www.rfc-editor.org/rfc/rfc793. Acesso em: 19 nov. 2025.