Manual do Projeto Integrador: "A Mini-Net"

Disciplina: Redes de Computadores – 2025/4

Curso: Engenharia de Software / Ciência da Computação / Sistemas de Informação / Inteligência Artificial

Tecnologia: Python 3 (Baterias inclusas: socket, threading, json)

1. Visão Geral

O objetivo deste Projeto Integrador é desmistificar o funcionamento da Internet. Em vez de apenas estudar a teoria das camadas OSI/TCP-IP, você irá implementar sua própria pilha de protocolos.

Neste projeto, desenvolveremos uma aplicação de chat (bate-papo). No entanto, não utilizaremos o protocolo TCP do sistema operacional para garantir a entrega das mensagens. Utilizaremos UDP (*User Datagram Protocol*) — um protocolo não confiável — e escreveremos, via código, todas as garantias de entrega, ordenação, roteamento e integridade que estudamos em sala.

O desafio: Fazer um chat funcionar perfeitamente sobre um canal de comunicação propositalmente defeituoso (que perde pacotes e corrompe dados).

2. Arquitetura do Sistema

O projeto segue a abordagem Top-Down (de cima para baixo). O sistema será composto por módulos que representam as camadas da rede. O encapsulamento deve ser respeitado: a Camada N só "conversa" com a Camada N-1.

Estrutura dos Dados (Encapsulamento):

Um dado enviado pelo chat seguirá a seguinte estrutura de "Bonecas Russas":

Quadro (Enlace) → Pacote (Rede) → Segmento (Transporte) → JSON (Aplicação)

3. Cronograma e Requisitos por Fase

O projeto é incremental. O código da Fase 1 será a base para a Fase 2, e assim por diante.

FASE 1: APLICAÇÃO E SOCKETS

Objetivo: Definir o protocolo de comunicação (a "linguagem" do chat) e estabelecer a conexão básica.

- **Requisitos:**
    1. Implementar uma arquitetura Cliente-Servidor (ou P2P).
    2. Definir um formato **JSON** para troca de mensagens (Ex.: campos type, sender, message, timestamp).
    3. O Cliente deve permitir a entrada de dados do usuário.
    4. O Servidor deve receber a mensagem do usuário e exibi-la formatada.

5. *Nota:* Nesta fase inicial, pode-se usar TCP apenas para validar o JSON, mas preparem-se para migrar para UDP na Fase 2.

FASE 2: TRANSPORTE E CONFIABILIDADE

Objetivo: Garantir que a mensagem chegue, mesmo que a rede falhe.

- **Mudança Crítica:** O socket deve ser alterado para SOCK_DGRAM (UDP). Deve-se utilizar o módulo simulador de erros (protocolo.py fornecido pelo professor) para enviar dados.
- **Requisitos:**
    1. **Stop-and-Wait:** Implementar o mecanismo de envio e espera.
    2. **ACKs:** O receptor deve enviar confirmações de recebimento.
    3. **Timeouts:** O emissor deve ter um temporizador. Se o ACK não chegar em X segundos, retransmitir.
    4. **Números de Sequência:** Implementar lógica para descartar duplicatas (bit 0 ou 1).

FASE 3: REDE E ROTEAMENTO

Objetivo: Introduzir endereçamento lógico e encaminhamento.

- **Requisitos:**
    1. Criar endereços virtuais (ex: "HOST_A", "SERVIDOR_PRIME").
    2. Implementar o campo **TTL (Time to Live)**.
    3. Criar um script roteador.py intermediário. O Cliente não envia mais diretamente ao Servidor, envia ao Roteador.
    4. O Roteador deve ler o destino virtual, consultar uma tabela estática e encaminhar para o IP/Porta corretos.

FASE 4: ENLACE E INTEGRIDADE

Objetivo: Detectar erros de bit (corrupção) causados pelo meio físico.

- **Requisitos:**
    1. Adicionar endereços MAC fictícios de origem e destino.
    2. **Checksum/CRC:** Antes de enviar, calcular o CRC32 do pacote e anexar ao quadro.
    3. **Verificação:** Ao receber, recalcular o CRC. Se divergir do recebido, descartar o quadro silenciosamente (simulando hardware).
    4. Demonstrar que a camada de Transporte (Fase 2) recupera essa perda através do timeout.

4. Especificações Técnicas e Restrições

- **Linguagem:** Python 3.8+.
- **Bibliotecas Permitidas:** Apenas bibliotecas padrão (socket, sys, time, json, random, struct, zlib, threading).
- **Proibições:**
    - É proibido usar TCP (SOCK_STREAM) nas fases 2, 3 e 4.
    - É proibido usar frameworks de alto nível (Flask, Django, Scapy) para a lógica do protocolo.
- **Simulação de Falhas:** O uso da função de "envio com ruído" (que simula perda e corrupção) é obrigatório na demonstração final.

5. Critérios de Avaliação

A nota será composta da seguinte forma (Total: 10,0 pontos):

Critério	Peso	Descrição
Funcionalidade Básica	3,0	O chat funciona? Mensagens vão e voltam?
Resiliência	3,0	O sistema recupera-se de pacotes perdidos e corrompidos? (Teste de estresse).
Implementação das Camadas	2,0	O código respeita o encapsulamento? (Classes separadas para Quadro, Pacote, Segmento).
Qualidade do Código	1,0	Código limpo, comentado e uso correto de Threads/Sockets.
Documentação/Logs	1,0	Logs no terminal mostrando o tráfego ("Enviando ACK", "Erro CRC", etc).

6. O que entregar?

O trabalho poderá ser realizado individualmente, em duplas ou trios.

A entrega final deve ser feita por e-mail ao Professor, marciano@ufg.br, com o [Link do Repositório GITHUB] contendo:

1. **Código Fonte:** Arquivos .py organizados (client.py, server.py, router.py, protocolo.py).
2. **Vídeo de Demonstração (3 a 5 min):** Gravura da tela mostrando:
    - O chat funcionando normalmente.
    - O log mostrando um erro de CRC/Perda e a retransmissão automática ocorrendo.
3. **Readme.md:** Pequeno manual de como rodar o seu projeto.

7. Dicas para o Sucesso

- **Logs são Vida**: Use print() coloridos para diferenciar as camadas. Ex: Vermelho para erros físicos, Amarelo para retransmissões de transporte, Verde para mensagens de aplicação.
- **Comece Simples**: Faça funcionar com texto simples antes de tentar enviar arquivos ou emojis.
- **Teste o Pior Caso**: Não teste seu código apenas quando a rede está perfeita. Configure a taxa de perda para 50% e veja se seu algoritmo de Stop-and-Wait realmente funciona.
- **Dividir para Conquistar**: Use classes separadas. Se você misturar a lógica de CRC com a lógica de JSON, seu código ficará impossível de debugar.

Bom trabalho! Lembrem-se: a Internet é um sistema confiável construído sobre componentes não confiáveis. Agora é a sua vez de construir essa confiabilidade.