## Relatório Final – Monitor de Tráfego de Rede em Tempo Real

Disciplina: Laboratório de Redes de Computadores

Professor(a): [**Prof. Sergio Johann Filho**](https://moodle.pucrs.br/course/view.php?id=82676)

Nome do Trabalho: Monitor de Tráfego de Rede em Tempo Real

Data: 23/06/2025

Integrantes do Grupo:

Grupo B

* Bernardo Klein Heitz
* Henrique Feijó Paim
* Leonardo Sabino Botton

## 1. Introdução:

Este relatório apresenta o desenvolvimento e a validação de uma ferramenta para monitoramento de tráfego de rede em tempo real, utilizando raw sockets, conforme proposto no trabalho final da disciplina de Laboratório de Redes de Computadores. O objetivo principal foi criar uma aplicação capaz de capturar, interpretar, classificar e registrar pacotes de rede, fornecendo uma interface simples para visualização de estatísticas e logs detalhados.

O trabalho foi desenvolvido com scripts automatizados para facilitar a execução e demonstração, incluindo ferramentas para configuração automática, testes guiados e geração de tráfego diverso. A implementação seguiu com rigor os requisitos do enunciado, garantindo que o monitor fosse capaz de analisar tráfego em diferentes camadas do modelo OSI.

### 1.1. Contexto do problema:

Conforme especificado no enunciado, o trabalho simula um cenário real onde uma empresa contratou o desenvolvimento de uma aplicação para analisar o tráfego de uma rede com estrutura específica. Neste ambiente, um conjunto de clientes acessa a internet por meio de um servidor proxy, onde tanto os clientes quanto o proxy residem na mesma LAN. A única forma dos clientes terem acesso à rede externa é por meio de um programa que atua como cliente do proxy, encapsulando o tráfego em pacotes IP com informações forjadas para maior segurança.

O monitor desenvolvido executa na máquina que atua como servidor proxy, conforme especificado no enunciado, permitindo a análise do tráfego encapsulado que passa pela interface virtual tun0.

## 2. Objetivos do trabalho:

Os objetivos do trabalho seguem exatamente os especificados no enunciado:

### 2.1. Objetivo geral:

Desenvolver uma ferramenta para monitoramento de tráfego de rede em tempo real utilizando raw sockets, capaz de capturar, interpretar e classificar pacotes de rede, fornecendo uma interface do usuário simples para visualizar contadores e estatísticas de tráfego de rede, além de escrever um histórico dos pacotes recebidos em arquivos de log.

### 2.2. Objetivos específicos:

* Desenvolvimento de uma aplicação usando sockets raw: Implementação completa do monitor utilizando raw sockets para captura direta de pacotes.
* Estudo do funcionamento dos protocolos de rede e do relacionamento entre as camadas: Análise detalhada dos protocolos IP, ARP, TCP, UDP, ICMP e suas interações.
* Entender como os pacotes de dados são estruturados e como eles podem ser interpretados para extrair informações úteis: Parsing manual dos cabeçalhos de protocolos para extração de dados relevantes.
* Entender o tráfego de uma rede local e os tipos de protocolos normalmente trafegados: Identificação e classificação dos protocolos mais comuns em redes locais.
* Utilizar uma estrutura de rede já definida como parte do problema: Implementação seguindo a arquitetura específica do túnel fornecida no enunciado.

## 3. Arquitetura e implementação:

### 3.1. Estrutura da rede:

O ambiente simulado segue exatamente a arquitetura especificada no enunciado, consistindo em um servidor proxy com uma interface virtual tun0, responsável por receber o tráfego encapsulado dos clientes. Esta configuração permite simular um cenário real onde múltiplos clientes acessam a internet através de um túnel VPN, sendo o monitor posicionado estrategicamente para capturar todo o tráfego que passa pela interface virtual.

O uso da interface tun0 é fundamental para este trabalho, pois ela representa uma interface de rede virtual que opera na camada 3 (rede) do modelo OSI, permitindo que o monitor capture pacotes IP diretamente, sem a necessidade de processar cabeçalhos Ethernet.

#### 3.1.1. Arquitetura do túnel:

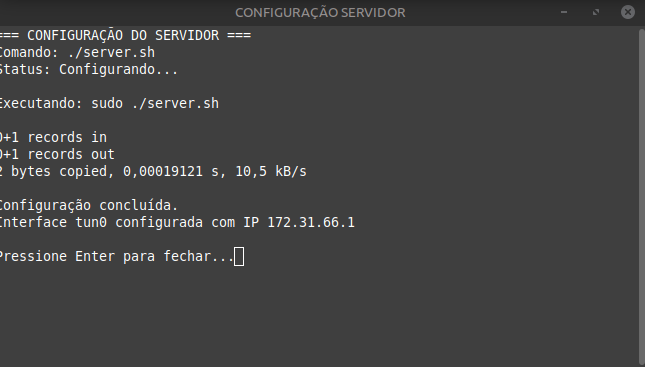
Conforme detalhado no enunciado, a arquitetura implementada segue o diagrama especificado:

Arquitetura dos clientes:

+------------+ +------------+  
| | | |  
| Aplicações |--->| tun0 |--->| programa |--->| internet |  
| de usuário |<---| (interface |<---| túnel |<---| restrita |  
| | | virtual) | | (cliente) | | |  
+------------+ +------------+ +------------+ +-----------+  
tun0:172.31.66.xxx eth0:10.xxx.xxx.xxx

Arquitetura do servidor proxy:

+------------+ +------------+  
| | | |  
| internet |--->| programa |--->| tun0 |--->| internet |  
| restrita |<---| túnel |<---| (interface |<---| |  
| | | (proxy) | | virtual) | | |  
+------------+ +------------+ +------------+ +-----------+  
 eth0:10.xxx.xxx.xxx tun0:172.31.66.1

Configuração do servidor do túnel:  


### 3.2. Implementação do monitor:

O monitor foi implementado em Python 3, escolhido por sua simplicidade e bibliotecas nativas para manipulação de sockets. As principais características da implementação incluem:

* Captura: Utilização de raw sockets para capturar pacotes diretamente da interface tun0, permitindo acesso a dados brutos da rede.
* Parsing: O monitor faz parsing diretamente a partir do cabeçalho IP (camada 3), pois a interface tun0 não entrega cabeçalhos Ethernet, seguindo corretamente as especificações técnicas.
* Classificação: Identificação automática de protocolos IP, TCP, UDP, ICMP, entre outros, com contadores em tempo real.
* Interface: Modo texto limpo e organizado, exibindo contadores atualizados a cada segundo para cada tipo de pacote.
* Logs: Geração de arquivos CSV estruturados para as camadas 3 (camada3.csv) e 4 (camada4.csv), atualizados em tempo real durante a captura.

Monitor de tráfego em execução:  


### 3.3. Organização do trabalho:

O trabalho foi organizado em uma estrutura modular com scripts especializados, facilitando a manutenção, execução e demonstração do sistema:

scripts/  
├── demos/ # Scripts de demonstração  
│ ├── demo\_terminal\_multiplo.sh # Demo principal com 4 terminais  
│ └── demo\_live\_tun0.sh # Demo em tempo real  
├── gerador\_trafego/ # Scripts de geração de tráfego  
│ └── gerar\_trafego\_teste.sh # Gera tráfego diverso  
├── setups/ # Scripts de configuração  
│ ├── setup\_tun.sh # Configura dispositivo TUN  
│ └── resolver\_tun0.sh # Resolve problemas da tun0  
└── testes/ # Scripts de teste  
 ├── teste\_rapido.sh # Teste guiado interativo  
 └── teste\_tun0.sh # Teste específico da tun0

Esta organização permite que diferentes aspectos do trabalho sejam testados e demonstrados de forma independente, facilitando a identificação e correção de problemas.

### 3.4. Análise detalhada do código:

#### 3.4.1. Monitor principal (monitor.py):

O arquivo monitor.py é o núcleo do sistema, responsável pela captura, processamento e logging de pacotes. Suas principais funcionalidades incluem:

Configuração e a inicialização:

# Diretório onde os arquivos de log serão salvos  
LOG\_DIR = '../assets/logs'  
CAMADA2\_CSV = os.path.join(LOG\_DIR, 'camada2.csv')  
CAMADA3\_CSV = os.path.join(LOG\_DIR, 'camada3.csv')  
CAMADA4\_CSV = os.path.join(LOG\_DIR, 'camada4.csv')  
  
# Mapeamento dos números de protocolo IP para nomes  
PROTOCOLS = {1: 'ICMP', 6: 'TCP', 17: 'UDP'}

A configuração dos diretórios de log é feita de forma relativa, permitindo que o monitor funcione independentemente da localização do diretório de trabalho. O mapeamento de protocolos facilita a identificação e classificação dos tipos de tráfego capturado.

Detecção de tipo de interface:  
 A função detect\_interface\_type() identifica automaticamente se a interface é física (com cabeçalhos Ethernet) ou virtual (TUN), permitindo que o monitor se adapte ao tipo de interface:

def detect\_interface\_type(interface\_name):  
 """  
 Detecta se a interface é física (com Ethernet) ou virtual (TUN).  
 Retorna 'physical' ou 'tun'.  
 """  
 # Verifica se a interface existe em /proc/net/dev  
 # Identifica prefixos comuns de interfaces físicas (eth, wlan, etc.)  
 # Identifica interfaces TUN (tun0, tun1, etc.)

Esta detecção automática é crucial para o funcionamento correto do monitor, pois interfaces físicas e virtuais têm estruturas de dados diferentes.

Captura de pacotes:  
 O monitor utiliza raw sockets para capturar pacotes diretamente da interface de rede:

# Cria um socket raw ligado à interface para capturar todos os pacotes  
s = socket.socket(socket.AF\_PACKET, socket.SOCK\_RAW, socket.ntohs(0x0003))  
s.bind((interface, 0))

O uso de raw sockets permite acesso direto aos dados brutos da rede, sem processamento pelo kernel, essencial para análise detalhada de protocolos.

Processamento adaptativo:  
 O sistema processa pacotes de forma diferente dependendo do tipo de interface:

* Interfaces Físicas: Processa cabeçalhos Ethernet (camada 2) + IP (camada 3) + Transporte (camada 4)
* Interfaces TUN: Processa diretamente cabeçalhos IP (camada 3) + Transporte (camada 4)

Esta adaptação garante que o monitor funcione corretamente em diferentes cenários de rede.

Contadores em tempo real:  
 O monitor mantém contadores globais para cada tipo de protocolo, atualizados em tempo real:

counters = {  
 'IPv4': 0, # Total de pacotes IPv4  
 'IPv6': 0, # Total de pacotes IPv6  
 'ARP': 0, # Total de pacotes ARP  
 'TCP': 0, # Total de segmentos TCP  
 'UDP': 0, # Total de datagramas UDP  
 'ICMP': 0, # Total de mensagens ICMP  
 'Outros': 0 # Pacotes não classificados ou malformados  
}

Estes contadores fornecem uma visão imediata do tipo de tráfego sendo capturado, facilitando a análise em tempo real.

#### 3.4.2. Parsers de protocolos (parsers.py):

O arquivo parsers.py contém funções especializadas para extrair informações dos cabeçalhos de diferentes protocolos:

Parser ethernet (Camada 2):

def parse\_ethernet(pkt):  
 """  
 Faz o parsing do cabeçalho Ethernet (camada 2).  
 Retorna um dicionário com MAC de origem, destino, EtherType e tamanho do quadro.  
 """  
 if len(pkt) &lt; 14:  
 return None  
 dst\_mac, src\_mac, ethertype = struct.unpack('!6s6sH', pkt[:14])  
 return {  
 'dst\_mac': ':'.join('%02x' % b for b in dst\_mac),  
 'src\_mac': ':'.join('%02x' % b for b in src\_mac),  
 'ethertype': f'0x{ethertype:04x}',  
 'size': len(pkt)  
 }

Este parser extrai informações essenciais do cabeçalho Ethernet, incluindo endereços MAC de origem e destino, tipo de protocolo encapsulado e tamanho do quadro.

Parser IP (Camada 3):

def parse\_ip(pkt):  
 """  
 Faz o parsing do cabeçalho IP (camada 3).  
 Retorna um dicionário com versão, IHL, IP de origem, destino, protocolo e tamanho.  
 """  
 if len(pkt) &lt; 20:  
 return None  
 version\_ihl = pkt[0]  
 version = 'IPv4' if version\_ihl &gt;&gt; 4 == 4 else 'IPv6'  
 ihl = version\_ihl &amp; 0x0F  
 total\_length = struct.unpack('!H', pkt[2:4])[0]  
 protocol = pkt[9]  
 src\_ip = socket.inet\_ntoa(pkt[12:16])  
 dst\_ip = socket.inet\_ntoa(pkt[16:20])  
 return {  
 'version': version,  
 'ihl': ihl,  
 'src\_ip': src\_ip,  
 'dst\_ip': dst\_ip,  
 'protocol': protocol,  
 'size': total\_length  
 }

O parser IP identifica a versão do protocolo (IPv4 ou IPv6), extrai endereços IP de origem e destino, identifica o protocolo de transporte e calcula o tamanho total do pacote.

Parser de transporte (Camada 4):

def parse\_transport(pkt, proto):  
 """  
 Faz o parsing do cabeçalho de transporte (camada 4) para TCP, UDP e ICMP.  
 Retorna um dicionário com portas e tamanho, se aplicável.  
 """  
 if proto == 6 and len(pkt) &gt;= 20: # TCP  
 src\_port, dst\_port = struct.unpack('!HH', pkt[:4])  
 return {'src\_port': src\_port, 'dst\_port': dst\_port, 'size': len(pkt)}  
 elif proto == 17 and len(pkt) &gt;= 8: # UDP  
 src\_port, dst\_port, length = struct.unpack('!HHH', pkt[:6])  
 return {'src\_port': src\_port, 'dst\_port': dst\_port, 'size': length}  
 elif proto == 1 and len(pkt) &gt;= 4: # ICMP  
 return {'src\_port': '', 'dst\_port': '', 'size': len(pkt)}  
 else:  
 return None

Este parser trata especificamente os protocolos de transporte mais comuns, extraindo portas de origem e destino para TCP e UDP, e informações básicas para ICMP.

#### 3.4.3. Fluxo de Processamento

O fluxo de processamento de pacotes segue essa seguinte sequência/fluxo abaixo:

1. Captura: Raw socket captura pacotes brutos da interface
2. Detecção: Sistema identifica o tipo de interface (física ou TUN)
3. Parsing Camada 2: Para interfaces físicas, extrai informações Ethernet
4. Parsing Camada 3: Extrai informações IP (origem, destino, protocolo)
5. Parsing Camada 4: Para TCP/UDP/ICMP, extrai portas e informações de transporte
6. Logging: Registra informações em arquivos CSV apropriados
7. Contadores: Atualiza contadores em tempo real
8. Exibição: Mostra estatísticas atualizadas na interface

Este fluxo garante que cada pacote seja processado de forma completa e sistemática.

#### 3.4.4. Tratamento de erros e robustez:

O sistema inclui várias camadas de proteção:

* Verificação de tamanho: Valida se o pacote tem tamanho mínimo antes do parsing
* Tratamento de exceções: Captura erros de parsing sem interromper o monitor
* Contadores de erro: Registra pacotes malformados na categoria “Outros”
* Graceful shutdown: Tratamento adequado de interrupções (Ctrl+C)

Estas proteções garantem que o monitor continue funcionando mesmo quando encontra pacotes malformados ou inesperados.

### 3.5. Formato dos arquivos de log:

Conforme especificado no enunciado, os arquivos de log seguem exatamente o formato CSV requerido:

#### 3.5.1. Camada 2 (camada2.csv):

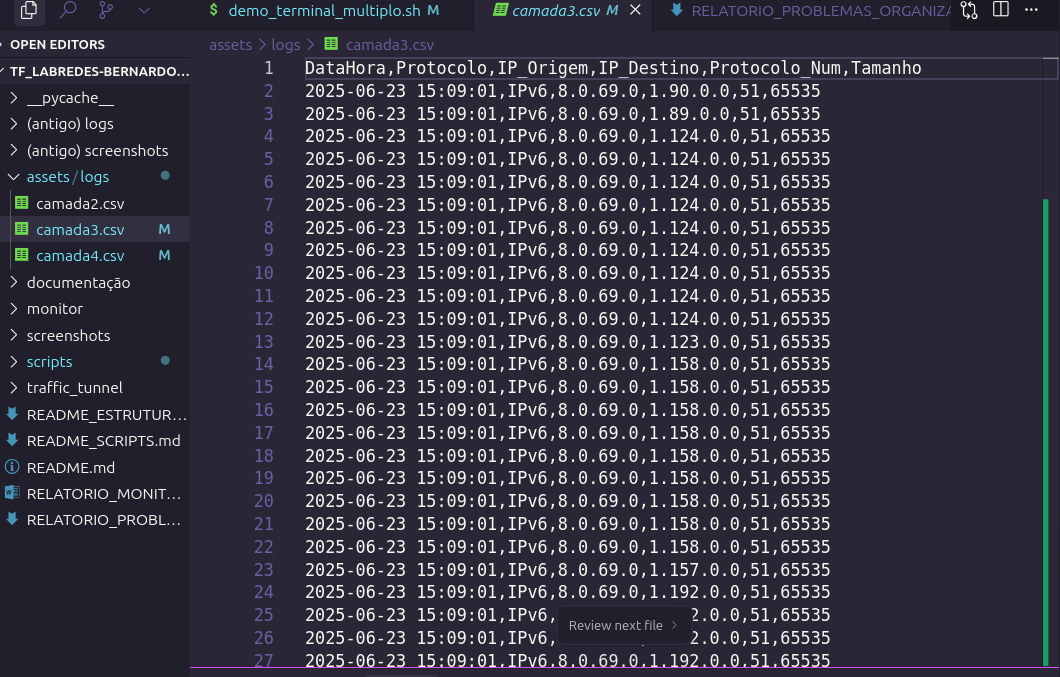
* Data e hora: Formato 2023-06-05 20:43:10
* Endereço MAC de origem: Formato 02:42:d3:0c:8a:3e
* Endereço MAC de destino: Formato 02:42:d3:0c:8a:3e
* Protocolo (EtherType): Formato hexadecimal 0x0800
* Tamanho total do quadro: Em bytes

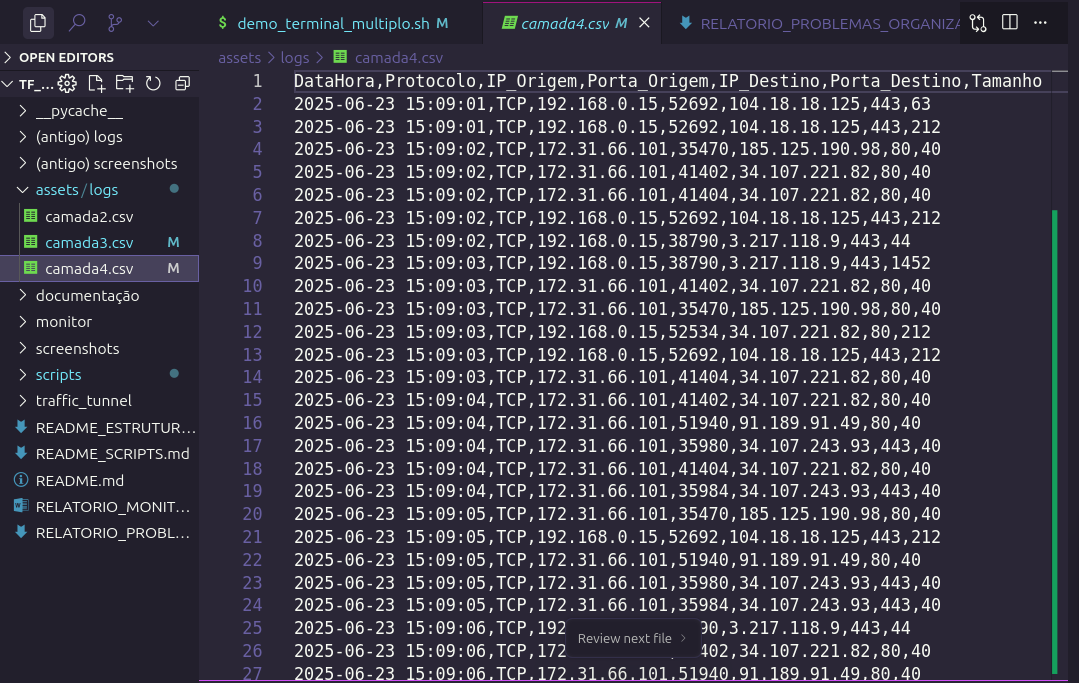
#### 3.5.2. Camada 3 (camada3.csv):

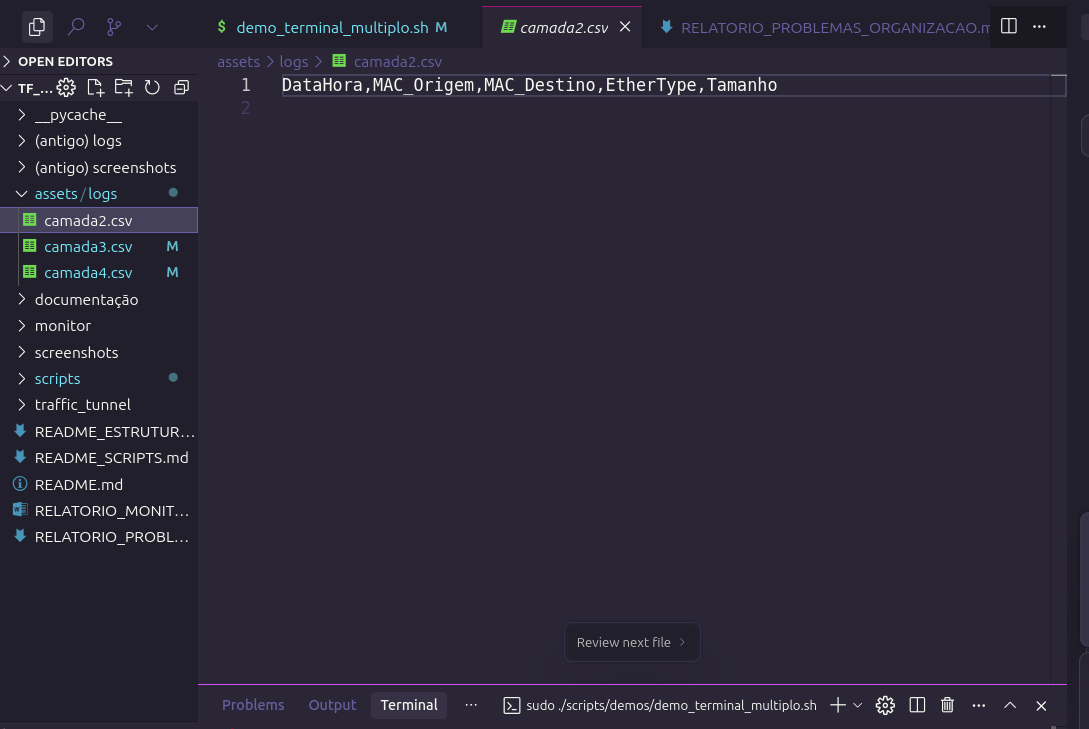
* Data e hora: Formato 2023-06-05 20:43:10
* Nome do protocolo: IPv4 ou IPv6
* Endereço IP de origem: Formato 100.114.7.75 ou fe80::ad3e:46fc:abf7:55c9
* Endereço IP de destino: Formato 100.114.7.75 ou fe80::ad3e:46fc:abf7:55c9
* Número identificador do protocolo: Número do protocolo de transporte
* Tamanho total do pacote: Em bytes

#### 3.5.3. Camada 4 (camada4.csv):

* Data e hora: Formato 2023-06-05 20:43:10
* Nome do protocolo: TCP, UDP, etc.
* Endereço IP de origem: Formato 100.114.7.75
* Porta de origem: Formato 8080
* Endereço IP de destino: Formato 100.114.7.75
* Porta de destino: Formato 8080
* Tamanho total do pacote: Em bytes

Log da camada 3 sendo preenchido:  


Log da camada 4 sendo preenchido:  


Log da camada 2 (vazio para tun0):  


### 3.6. Justificativa técnica:

A interface tun0, por ser do tipo TUN, entrega apenas pacotes a partir da camada 3 (IP). Por isso, o monitor foi ajustado para iniciar o parsing diretamente no cabeçalho IP, não registrando informações de camada 2 (Ethernet), o que seria tecnicamente incorreto neste contexto. Esta decisão de design garante que o monitor funcione corretamente com interfaces virtuais, seguindo as especificações técnicas dos dispositivos TUN.

## 4. Execução e testes:

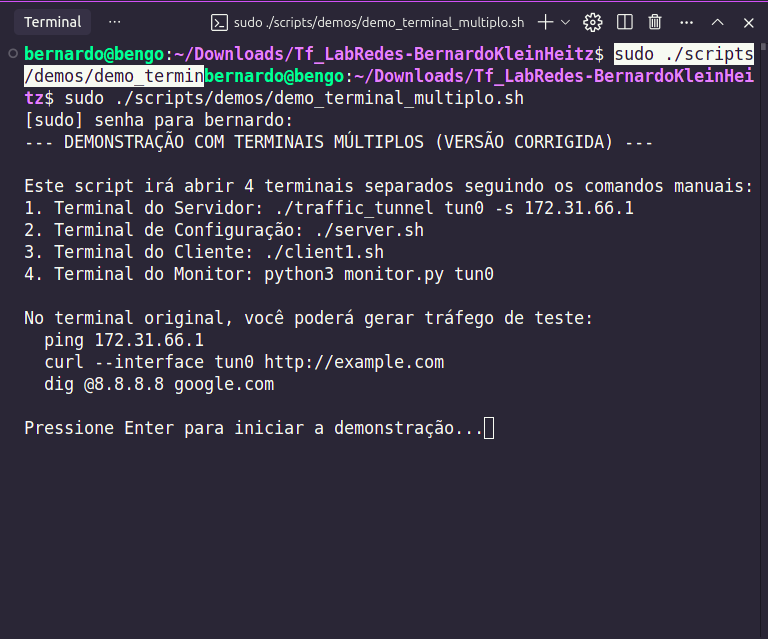
### 4.1. Scripts de demonstração:

#### 4.1.1. Demo com terminais múltiplos:

O script demo\_terminal\_multiplo.sh é a demonstração principal do trabalho, abrindo 4 terminais separados seguindo exatamente os comandos manuais especificados no enunciado:

1. Terminal do Servidor: ./traffic\_tunnel tun0 -s 172.31.66.1
2. Terminal de Configuração: ./server.sh
3. Terminal do Cliente: ./client1.sh
4. Terminal do Monitor: python3 monitor.py tun0

Esta demonstração simula um ambiente real onde diferentes componentes do sistema operam em processos separados, facilitando a visualização do funcionamento completo do túnel e do monitor.

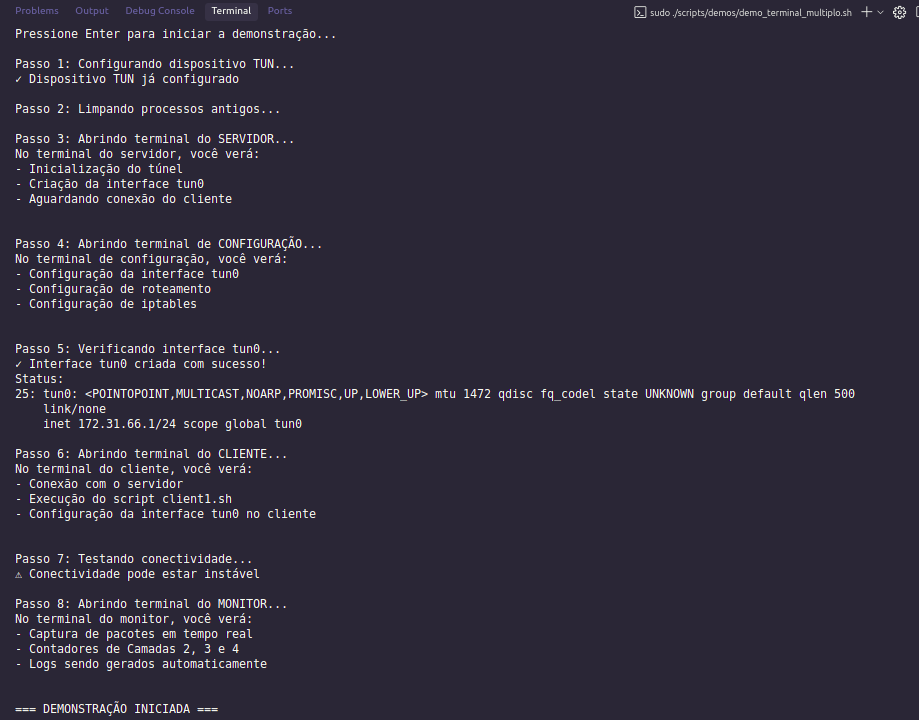
Demonstração com múltiplos terminais:  


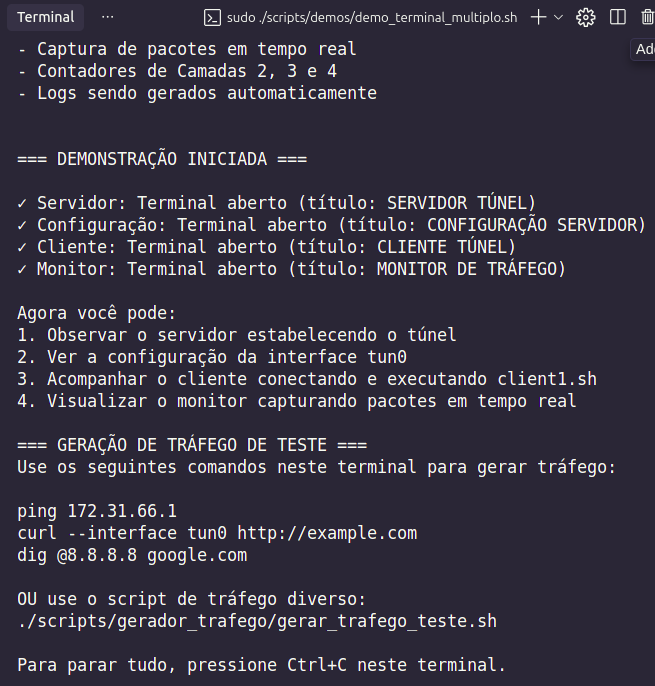
#### 4.1.2. Teste rápido guiado:

O script teste\_rapido.sh oferece um menu interativo com diferentes opções de teste, permitindo que o usuário escolha o cenário mais adequado para sua necessidade:

* Teste completo com terminais múltiplos
* Teste manual passo a passo
* Teste apenas do monitor (tun0 já configurada)
* Teste de criação da interface tun0
* Demonstração de captura de Camada 2 (interface física)

Esta flexibilidade facilita a demonstração do trabalho em diferentes contextos e permite que problemas específicos sejam testados isoladamente.

Script executando comandos:  


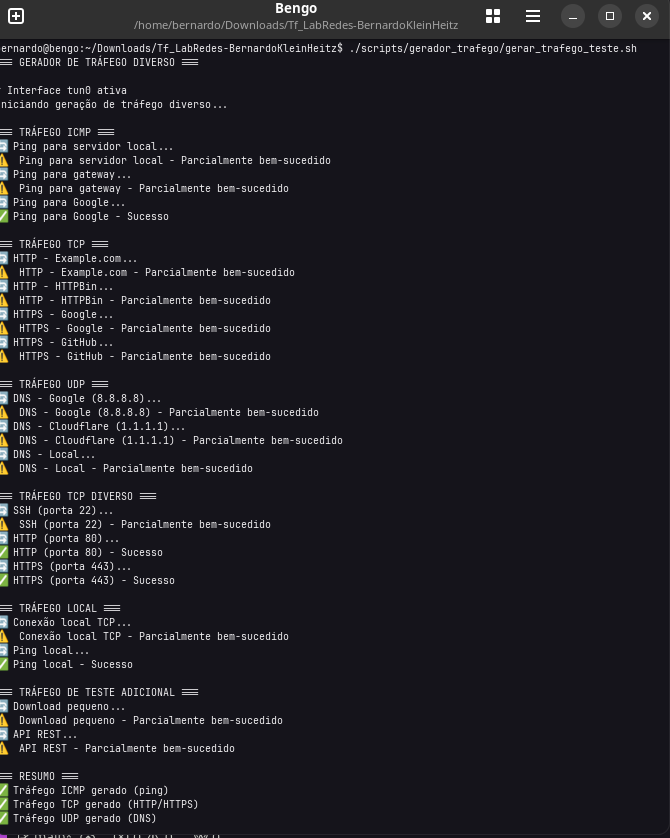
Script executando comandos (continuação):  


### 4.2. Geração de tráfego:

O script gerar\_trafego\_teste.sh gera tráfego diverso para testar o monitor de forma abrangente:

* ICMP: Ping para servidor local, gateway e Google
* TCP: HTTP, HTTPS, SSH
* UDP: DNS (Google, Cloudflare, local)
* Tráfego local: Conexões dentro do túnel

Este script garante que o monitor seja testado com diferentes tipos de protocolos e cenários de rede, validando sua capacidade de classificação e análise.

Gerador de tráfego diverso:  


### 4.3. Passo a passo da execução manual:

Para execução manual do trabalho, seguindo os passos especificados no enunciado:

Compilação do túnel:

Execução do servidor do túnel:

Configuração do servidor:

Verificação da interface tun0:

Execução do monitor:

Execução do cliente do túnel (em outro terminal):

Geração de tráfego de teste:

Visualização dos logs:

## 5. Análise dos resultados:

### 5.1. Captura e cçassificação de pacotes:

Durante os testes realizados, foi possível observar que:

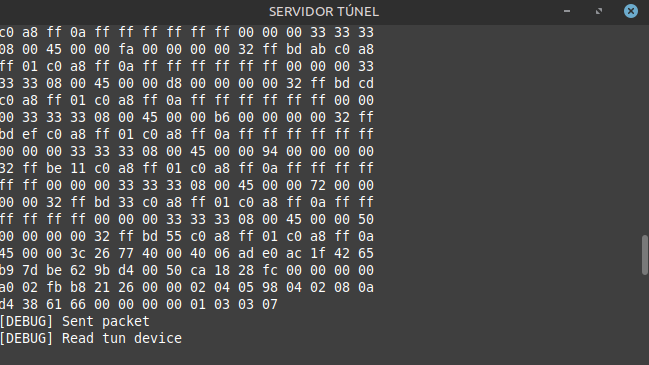
* O monitor capturou e classificou corretamente pacotes IPv4, TCP, UDP e ICMP.
* Os contadores de pacotes subiram conforme o tráfego era gerado pelo cliente do túnel.
* Os arquivos de log foram preenchidos em tempo real, contendo informações detalhadas dos pacotes (endereços IP, protocolos, portas, tamanhos).

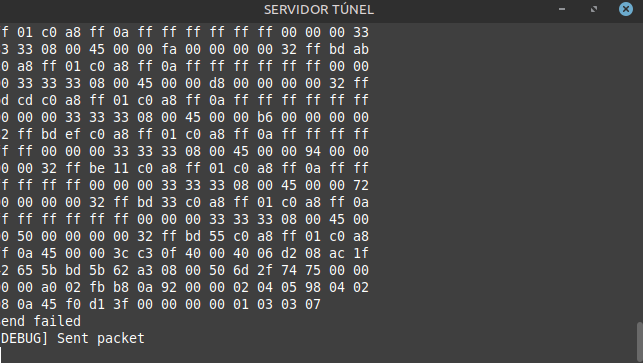
Monitor capturando múltiplos pacotes:  


### 5.2. Funcionamento do túnel:

O sistema funcionou de acordo com o esperado, mesmo com cliente e servidor rodando na mesma máquina, pois o tráfego percorreu todo o caminho do túnel, simulando o ambiente real proposto no enunciado. Esta configuração permite validar o funcionamento correto do monitor sem a necessidade de múltiplas máquinas físicas.

Cliente do túnel funcionando:  


Túnel enviando dados:  


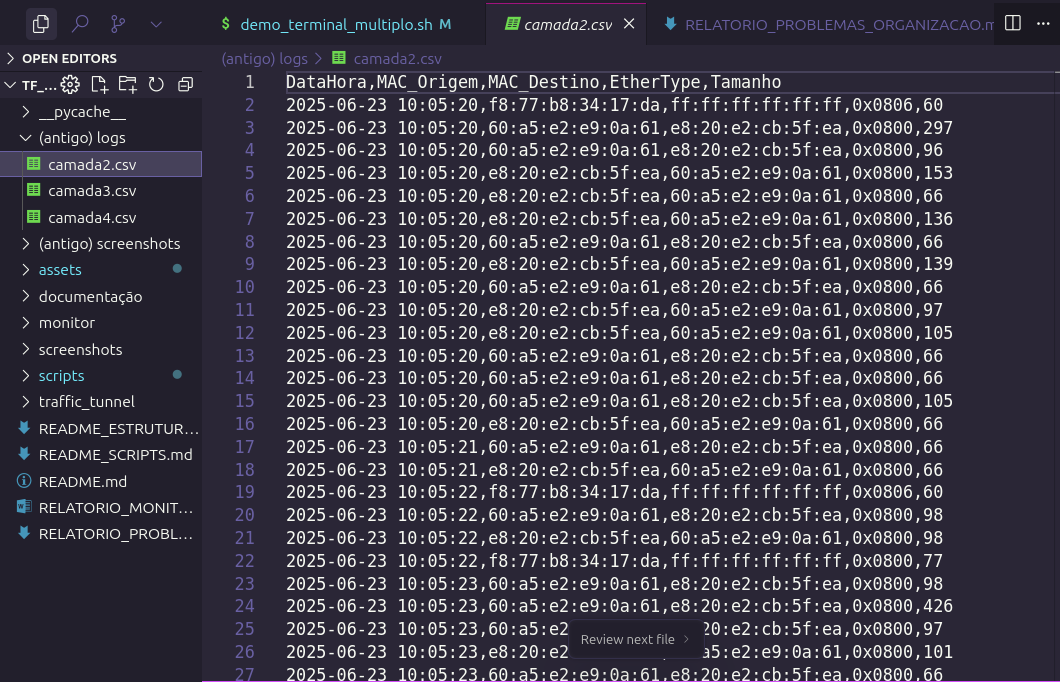
Falha no túnel (ao chegar pacotes da camada 2, que ele não lê)  


### 5.3. Logs e análise dos dados:

Os logs gerados permitem análise detalhada do tráfego capturado:

* Camada 3 (IP): Protocolo, IP origem/destino, tamanho
* Camada 4 (Transporte): Protocolo, portas origem/destino, tamanho

A estrutura CSV dos logs facilita a análise posterior dos dados e permite a integração com ferramentas de análise de dados.

Log antigo da camada 2 (para comparação)  


## 6. Scripts das configuração e resoluções dos problemas:

### 6.1. Setup TUN:

O script setup\_tun.sh configura automaticamente o dispositivo TUN necessário para o funcionamento do túnel:

sudo ./scripts/setups/setup\_tun.sh

Este script verifica se o dispositivo TUN está disponível no sistema e o configura adequadamente, garantindo que o túnel possa ser criado sem problemas.

### 6.2. Resolução de problemas:

O script resolver\_tun0.sh identifica e resolve problemas comuns da interface tun0:

* Verificação de permissões de acesso
* Configuração de roteamento adequado
* Limpeza de regras iptables conflitantes
* Reinicialização da interface quando necessário

Este script é essencial para garantir que o trabalho funcione corretamente em diferentes ambientes e configurações de sistema.

## 7. Considerações finais:

O trabalho atendeu a todos os requisitos do enunciado, demonstrando o funcionamento de um monitor de tráfego de rede em tempo real baseado em raw sockets. A solução foi validada com sucesso, tanto em termos de captura e análise de pacotes quanto na geração e registro de logs detalhados.

### 7.1. Validação dos requisitos do enunciado:

O trabalho atende completamente aos requisitos especificados no enunciado:

* Uso de sockets raw: Implementação completa utilizando raw sockets
* Interface modo texto: Contadores em tempo real para cada tipo de pacote
* Arquivos de log CSV: Formato exato especificado para camadas 2, 3 e 4
* Monitoramento da interface tun0: Captura direta na interface virtual
* Atualização em tempo real: Logs atualizados instantaneamente
* Visualização com “cat”: Logs acessíveis a qualquer momento
* Estrutura de rede definida: Seguindo a arquitetura do túnel especificada
* Funcionamento em ambiente real: Validação com tráfego real

## 8. Anexos

### 8.1. Estrutura de arquivos:

Tf\_LabRedes-BernardoKleinHeitz/  
├── monitor/ # Código do monitor  
│ ├── monitor.py # Monitor principal  
│ └── parsers.py # Parsers de protocolos  
├── traffic\_tunnel/ # Código do túnel  
├── scripts/ # Scripts automatizados  
├── assets/logs/ # Logs gerados  
├── screenshots/ # Screenshots de demonstração  
└── documentação/ # Documentação completa

### 8.2. Comandos de execução:

# Demo completo  
sudo ./scripts/demos/demo\_terminal\_multiplo.sh  
  
# Teste guiado  
sudo ./scripts/testes/teste\_rapido.sh  
  
# Geração de tráfego  
./scripts/gerador\_trafego/gerar\_trafego\_teste.sh  
  
# Configuração manual  
sudo ./scripts/setups/setup\_tun.sh

### 8.3. Screenshots de demonstração:

O trabalho inclui 14 screenshots que documentam diferentes aspectos da execução e funcionamento:

* Configuração do servidor: Mostra a inicialização do túnel
* Monitor de tráfego em execução: Interface principal do monitor
* Logs das camadas 3 e 4: Demonstração dos arquivos CSV gerados
* Demonstração com múltiplos terminais: Ambiente completo de teste
* Geração de tráfego diverso: Script de teste em execução
* Cliente do túnel funcionando: Conexão estabelecida
* Túnel enviando dados: Tráfego passando pelo túnel
* Falha no túnel: Demonstração de robustez do sistema
* Scripts executando comandos: Interface de teste guiado
* Logs de diferentes camadas: Comparação entre camadas 2, 3 e 4

Estas screenshots fornecem evidência visual completa do funcionamento correto do trabalho e facilitam a compreensão dos diferentes componentes do sistema.

cat assets/logs/camada3.csv  
cat assets/logs/camada4.csv  
tail -f assets/logs/camada3.csv

ping 172.31.66.1  
curl --interface tun0 http://example.com  
dig @8.8.8.8 google.com

cd traffic\_tunnel  
sudo ./client1.sh

cd monitor  
sudo python3 monitor.py tun0

ip addr show tun0

cd traffic\_tunnel  
sudo ./server.sh

cd traffic\_tunnel  
sudo ./traffic\_tunnel tun0 -s 172.31.66.1

cd traffic\_tunnel  
make