Conteúdo

1 Introdução

Este relatório apresenta os procedimentos, análises e resultados obtidos no desenvolvimento do Trabalho Prático II da disciplina Redes TCP/IP. O objetivo principal do trabalho é aprofundar o entendimento prático sobre protocolos fundamentais da pilha TCP/IP, com foco nos utilitários *ping* e *traceroute*, bem como nas medições de desempenho da comunicação via protocolo UDP em uma rede local real. O Experimento 1 visa explorar, por meio de pesquisa e testes empíricos, o comportamento do RTT e a variação de rotas em diferentes horários e dias, utilizando o traceroute com endereços IPv4. Já o Experimento 2 tem como finalidade implementar e avaliar o desempenho de um sistema de comunicação UDP, medindo latência e vazão sob diferentes condições. Ao final, espera-se que os experimentos contribuam para a compreensão dos conceitos teóricos por meio da aplicação prática e da análise crítica dos resultados obtidos.

2 Experimento 1

Existem dois utilitários de rede muito populares, desenvolvidos originalmente no ambiente Unix, mas hoje também disponíveis nos outros sistemas operacionais, que servem para determinar se um endereço IP específico está ativo na rede, o valor aproximado do RTT (roundtrip time) até o mesmo e a rota para alcançá-lo: o *ping* e o *traceroute*.

a) Qual é a diferença entre o uso do ping e do traceroute? Como o ping é implementado (em detalhes, que mecanismos e protocolos da família TCP/IP ele usa para funcionar)? Como o traceroute é implementado (em detalhes, que mecanismos e protocolos da família TCP/IP ele usa para funcionar)?

A priori, é mister destacar que o *ping* verifica a acessibilidade de um host remoto e mede a latência entre origem e destino, ao passo que o *traceroute*, por sua vez, determina o caminho completo que os pacotes percorrem até alcançar esse destino, revelando os roteadores intermediários.

O comando *ping* foi inspirado nos princípios do sonar, emitindo uma "onda" na forma de um pacote *ICMP Echo Request* e aguardando um *ICMP Echo Reply* como resposta. Seu funcionamento, portanto, está baseado no protocolo IP e no ICMP, onde o protocolo ICMP atua como um mecanismo auxiliar ao IP, fornecendo mensagens de controle. O *ping* não utiliza as camadas de transporte TCP ou UDP, operando diretamente sobre a camada de rede. O tempo de resposta (RTT — *Round Trip Time*) é calculado a partir do intervalo entre o envio do pacote e a recepção da resposta correspondente. Essa simplicidade permite que a ferramenta seja utilizada para diagnósticos rápidos, como detecção de perda de pacotes e medições de latência. [**?** , p. 465–467]

Por outro lado, o *traceroute* (ou *tracert* em sistemas Windows) explora o campo TTL (Time To Live) presente nos pacotes IP para descobrir o caminho que os dados percorrem até o destino. O TTL é um valor numérico que representa o número máximo de saltos (*hops*) que um pacote pode realizar antes de ser descartado. Ele envia pacotes com valores TTL começando em 1 e incrementando esse valor a cada nova tentativa. Quando um roteador recebe um pacote com TTL = 1, ele descarta o pacote e envia de volta uma mensagem *ICMP Time Exceeded* para o emissor. Esse comportamento é padronizado pela RFC 792. Ao receber essas mensagens, o *traceroute* consegue identificar o endereço IP de cada roteador intermediário. [**?** , p. 465–467] [**?**]

A implementação do *traceroute* varia conforme o sistema operacional. Em sistemas Unix e Linux, os pacotes enviados geralmente são UDP com destino a portas altas (acima de 33434). O destino, ao receber esses pacotes, responderá com uma mensagem *ICMP Port Unreachable* caso a porta não esteja aberta, permitindo ao *traceroute* identificar o fim do percurso. Já em sistemas Windows, utiliza-se diretamente *ICMP Echo Requests* com TTLs incrementais, dispensando o uso de UDP. [**?**]

Além dos livros clássicos, pesquisas contemporâneas também investigam formas mais eficientes de varredura de caminhos de rede. Um exemplo é o estudo de Beverly [**?**], que propõe uma varredura paralela e randômica utilizando o Yarrp (*Yelling at Random Routers Progressively*), uma evolução do *traceroute* tradicional. O trabalho ressalta que a coleta de rotas é limitada por fatores como balanceamento de carga, políticas de roteamento e bloqueios a pacotes ICMP, mas continua sendo essencial para mapeamentos ativos da topologia da Internet.

Destarte, *ping* e *traceroute* são ferramentas complementares: enquanto o primeiro confirma se um host está alcançável e mede o tempo de resposta, o segundo detalha o caminho tomado até esse host. Ambas utilizam o ICMP, mas de formas distintas, aproveitando-se de diferentes campos e mensagens desse protocolo para cumprir seus objetivos.

b) Utilizando o comando traceroute, com a opção de uso apenas de endereços IPv4, deve-se levantar o RTT e a rota na Internet para cada um dos dois destinos abaixo. Para isso, para cada um dos destinos, deverão ser realizadas medições em três períodos diferentes de um dia útil da semana: 6-12 h, 12-18 h e 18-24 h. Em cada um dos períodos, devem ser feitas 10 medições de rota e RTT, sendo que a média dos 10 valores medidos de RTT representará o valor do RTT daquele período. O mesmo procedimento deve ser repetido para um dia de fim de semana. Para cada destino, deverão ser apresentadas, em tabelas, todos os valores de RTT obtidos e as médias respectivas, sendo que os resultados de RTT médio por período do dia deverão ser plotados em um gráfico tipo histograma. Além disso, as rotas obtidas em cada período deverão ser listadas, destacando possíveis variações nas mesmas.

- DESTINO 1: www.nic.br
- DESTINO 2: www.ufrgs.br
- DESTINO 3: www.ufal.br
- DESTINO 4: www.ufms.br

c) Baseado nas "medições" realizadas, deve-se verificar: houve variação do valor do RTT conforme o horário ou o dia? Deve-se explicar em detalhes o que ocorreu e buscar justificativas para o comportamento observado.

d) Baseado nas "medições" realizadas, deve-se verificar: houve variação de rotas conforme o horário ou o dia? Deve-se explicar em detalhes o que ocorreu e buscar justificativas para o comportamento observado.

3 Experimento 2

3.1 Enunciado

Este experimento consiste na realização do Exercício 48, letras (a) e (b), do Capítulo 5 do livro-texto adotado, com foco exclusivo no protocolo UDP. O objetivo é medir a latência e a vazão (throughput) na comunicação entre dois hosts (Host A e Host B) em uma rede local.

Para cada um dos tamanhos de mensagem especificados:

- **Teste Básico:** Enviar uma mensagem de tamanho determinado do Host A para o Host B. O Host B deve refletir (retornar) a mensagem de volta para o Host A.
- **Repetições:** Esta sequência de envio e reflexão deve ser repetida 100.000 vezes para cada tamanho de mensagem.
- **Latência:** A latência é calculada como o tempo total decorrido para as 100.000 repetições, dividido pelo número de repetições bem-sucedidas.
- **Vazão (Throughput):** A vazão é calculada como o tamanho da mensagem (em bits) dividido pela latência média (em segundos) de uma única reflexão. (Vazão = (Tamanho da Mensagem * 8) / Latência).
- **Trials:** Cada teste completo (100.000 repetições) para um dado tamanho de mensagem deve ser executado três vezes (trials). O valor final de latência e vazão para esse tamanho de mensagem será a média aritmética dos resultados obtidos nos três trials.

Os resultados devem ser apresentados em tabelas e gráficos, tanto para latência quanto para vazão, em função do tamanho da mensagem. Adicionalmente, uma discussão detalhada dos resultados e justificativas para os comportamentos observados é requerida.

Os tamanhos de mensagem são divididos em duas categorias:

- Parte (a): Mensagens pequenas, variando de 1 a 1000 bytes.
- Parte (b): Mensagens grandes, variando de 1KB a 32KB.

3.2 Detalhamento do Ambiente do Experimento

Os testes foram conduzidos utilizando dois hosts fisicamente distintos, denominados Host A (testador) e Host B (refletor), conectados em uma rede local (WLAN). Conforme as diretrizes, não foram utilizadas máquinas virtuais no mesmo host físico, nem comunicação via Internet.

3.2.1 Host A (Testador)

- **Sistema Operacional:** 22.04.2 LTS
- **Processador:** Intel® Core™ i7-7700 CPU @ 3.60GHz × 8
- **Memória RAM:** 16,0 GB
- **Disco:** 35,0 GB

3.2.2 Host B (Refletor)

- **Sistema Operacional:** Ubuntu 24.04.2 LTS
- **Processador:** AMD Ryzen™ 7 5700U with Radeon™ Graphics × 16
- **Memória RAM:** 8,0 GB
- **Disco:** 37,0 GB

3.2.3 Rede Local (Ethernet)

- **Padrão Ethernet:** IEEE 802.3u (Fast Ethernet)
- **Taxa de Conexão Nominal:** 100 Mbps
- **Switch/Roteador:** D-Link DES-1016D

3.2.4 Software Utilizado

- **Linguagem de Programação:** C.
- **Compilador:** GCC (GNU Compiler Collection).
- **Sistema de Geração de Gráficos:** Matplotlib em Python

3.3 Código Fonte Produzido

Foram desenvolvidos dois programas em C: um refletor UDP (`reflector_udp.c`) para o Host B e um testador UDP (`tester_udp.c`) para o Host A.

3.3.1 Refletor UDP (`reflector_udp.c`)

O programa `reflector_udp.c` é executado no Host B. Sua função é escutar por datagramas UDP em uma porta específica (definida como 8080). Ao receber um datagrama de qualquer cliente, o refletor simplesmente o envia de volta para o endereço e porta do remetente, sem modificar o conteúdo.

Principais características do refletor:

- Criação de um socket UDP.
- Vinculação (bind) do socket ao endereço IP da máquina e à porta 8080, aceitando conexões de qualquer interface de rede (`INADDR_ANY`).
- Loop infinito para receber e reenviar mensagens.
- Utiliza `recvfrom()` para receber dados e informações do remetente.
- Utiliza `sendto()` para enviar os dados recebidos de volta ao remetente.
- O tamanho do buffer de recepção é `MAX_BUFFER_SIZE` (65536 bytes), suficiente para o maior datagrama UDP teórico.

Abaixo, o código fonte do `reflector_udp.c`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 8080
#define MAX_BUFFER_SIZE 65536 // Suficiente para o maior datagrama UDP (teorico)

int main() {
    int sockfd;
    struct sockaddr_in servaddr, cliaddr;
    char buffer[MAX_BUFFER_SIZE];
    socklen_t len;
    ssize_t n; // Usar ssize_t para o retorno de recvfrom/sendto

    // Criando o socket UDP
```

```c
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));
    memset(&cliaddr, 0, sizeof(cliaddr));

    // Configurando o endereco do servidor
    servaddr.sin_family = AF_INET; // IPv4
    servaddr.sin_addr.s_addr = INADDR_ANY; // Aceita conexoes de qualquer IP
    servaddr.sin_port = htons(PORT);

    // Bind do socket com o endereco do servidor
    if (bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0) {
        perror("bind failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("Refletor UDP escutando na porta %d...\n", PORT);

    while (1) {
        len = sizeof(cliaddr);
        // Recebe a mensagem do cliente
        n = recvfrom(sockfd, (char *)buffer, MAX_BUFFER_SIZE,
                     0, (struct sockaddr *)&cliaddr, &len);
        if (n < 0) {
            perror("recvfrom error");
            continue;
        }

        // Envia a mensagem de volta para o cliente
        // Usar 'n' como tamanho, que eh o numero de bytes recebidos
        if (sendto(sockfd, (const char *)buffer, n,
                   0, (const struct sockaddr *)&cliaddr, len) < 0) {
            perror("sendto error");
            // Continuar mesmo se sendto falhar para uma solicitacao especifica
        }
    }

    close(sockfd); // Em teoria, nunca alcancado neste loop infinito
    return 0;
}
```

Listing 1: Código fonte do Refletor UDP (reflector_udp.c)

3.3.2 Testador UDP (`tester_udp.c`)

O programa `tester_udp.c` é executado no Host A. Ele é responsável por conduzir os testes de latência e vazão, enviando mensagens de diferentes tamanhos para o refletor no Host B e medindo o tempo de ida e volta.

Principais características do testador:

- Recebe o endereço IP do Host B como argumento de linha de comando.
- Define constantes para o número de repetições por teste (`NUM_REPETITIONS = 100000`) e o número de trials por tamanho de mensagem (`NUM_TRIALS = 3`).
- Testa duas faixas de tamanhos de mensagem:
  - Pequenas: 1, 100, 200, ..., 1000 bytes.

  - Grandes: 1KB, 2KB, ..., 32KB (convertidos para bytes).

- Para cada tamanho de mensagem e cada trial:
  - Preenche um buffer de envio com dados.
  - Mede o tempo total para `NUM_REPETITIONS` operações de envio (`sendto`) e recebimento (`recvfrom`).
  - Utiliza `clock_gettime(CLOCK_MONOTONIC, ...)` para medição de tempo de alta precisão.
  - Configura um timeout de 1 segundo para a operação `recvfrom` usando `setsockopt` com `SO_RCVTIMEO` para evitar bloqueio indefinido em caso de perda de pacotes.
  - Conta o número de repetições bem-sucedidas (onde a mensagem é enviada e uma resposta de tamanho correto é recebida).
- Calcula a latência média e a vazão média para cada trial.
- Calcula a média final da latência e da vazão a partir dos resultados dos `NUM_TRIALS` trials.
- Imprime os resultados formatados na saída padrão.
- Salva os dados brutos de latência e vazão em quatro arquivos de texto: `data_latency_small.txt`, `data_throughput_small.txt`, `data_latency_large.txt`, e `data_throughput_large.txt`.

Abaixo, o código fonte do `tester_udp.c`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>   // Para clock_gettime
#include <errno.h>  // Para errno

#define SERVER_PORT 8080
#define NUM_REPETITIONS 100000
#define NUM_TRIALS 3
#define MAX_PAYLOAD_SIZE 32768 // 32KB
#define RECV_BUFFER_SIZE (MAX_PAYLOAD_SIZE + 100) // Um pouco maior para seguranca

// Funcao para obter tempo em segundos com alta precisao
double get_time_sec() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec / 1e9;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Uso: %s <IP_DO_HOST_B>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *server_ip = argv[1];

    int sockfd;
    struct sockaddr_in servaddr;
    char *send_buffer;
    char recv_buffer[RECV_BUFFER_SIZE];

    send_buffer = (char*) malloc(MAX_PAYLOAD_SIZE);
    if (!send_buffer) {
        perror("malloc send_buffer failed");
```

```
39          exit(EXIT_FAILURE);
40      }
41
42      // Arquivos para dados brutos
43      FILE *fp_latency_small, *fp_throughput_small;
44      FILE *fp_latency_large, *fp_throughput_large;
45
46      fp_latency_small = fopen("data_latency_small.txt", "w");
47      fp_throughput_small = fopen("data_throughput_small.txt", "w");
48      fp_latency_large = fopen("data_latency_large.txt", "w");
49      fp_throughput_large = fopen("data_throughput_large.txt", "w");
50
51      if (!fp_latency_small || !fp_throughput_small || !fp_latency_large ||
            !fp_throughput_large) {
52          perror("Erro ao abrir arquivos de dados para escrita");
53          if(fp_latency_small) fclose(fp_latency_small);
54          if(fp_throughput_small) fclose(fp_throughput_small);
55          if(fp_latency_large) fclose(fp_latency_large);
56          if(fp_throughput_large) fclose(fp_throughput_large);
57          free(send_buffer);
58          exit(EXIT_FAILURE);
59      }
60
61      fprintf(fp_latency_small, "# Tam_Msg_Bytes Latencia_s\n");
62      fprintf(fp_throughput_small, "# Tam_Msg_Bytes Vazao_bps\n");
63      fprintf(fp_latency_large, "# Tam_Msg_Bytes Latencia_s\n");
64      fprintf(fp_throughput_large, "# Tam_Msg_Bytes Vazao_bps\n");
65
66
67      if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
68          perror("socket creation failed");
69          // Lidar com fechamento de arquivos e free
70          fclose(fp_latency_small); fclose(fp_throughput_small);
71          fclose(fp_latency_large); fclose(fp_throughput_large);
72          free(send_buffer);
73          exit(EXIT_FAILURE);
74      }
75
76      struct timeval tv;
77      tv.tv_sec = 1; // Timeout de 1 segundo para recvfrom
78      tv.tv_usec = 0;
79      if (setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, (const char*)&tv, sizeof tv) <
            0) {
80          perror("setsockopt SO_RCVTIMEO failed");
81          close(sockfd);
82          fclose(fp_latency_small); fclose(fp_throughput_small);
83          fclose(fp_latency_large); fclose(fp_throughput_large);
84          free(send_buffer);
85          exit(EXIT_FAILURE);
86      }
87
88      memset(&servaddr, 0, sizeof(servaddr));
89      servaddr.sin_family = AF_INET;
90      servaddr.sin_port = htons(SERVER_PORT);
91      if (inet_pton(AF_INET, server_ip, &servaddr.sin_addr) <= 0) {
92          perror("inet_pton error for IP address");
93          close(sockfd);
94          fclose(fp_latency_small); fclose(fp_throughput_small);
95          fclose(fp_latency_large); fclose(fp_throughput_large);
96          free(send_buffer);
97          exit(EXIT_FAILURE);
```

```
98      }
99
100     int msg_sizes_a[] = {1, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
101     int num_sizes_a = sizeof(msg_sizes_a) / sizeof(msg_sizes_a[0]);
102
103     int msg_sizes_b_kb[32];
104     for(int i=0; i<32; ++i) msg_sizes_b_kb[i] = i + 1; // 1KB to 32KB
105     int num_sizes_b = 32;
106     int msg_sizes_b[num_sizes_b];
107     for (int i = 0; i < num_sizes_b; ++i) {
108         msg_sizes_b[i] = msg_sizes_b_kb[i] * 1024;
109     }
110
111     printf("Iniciando testes UDP para o servidor %s...\n", server_ip);
112
113     // === PARTE (a): Mensagens Pequenas (1-1000 Bytes) ===
114     printf("\nResultados para Parte (a) (UDP - Latencia e Vazao - Mensagens
            Pequenas):\n");
115     printf("-------------------------------------------------------------\n");
116     printf("| Tam. Msg (Bytes) | Latencia Media (s) | Vazao Media (bps) |\n");
117     printf("|------------------|--------------------|-------------------|\n");
118
119     for (int i = 0; i < num_sizes_a; ++i) {
120         int current_msg_size = msg_sizes_a[i];
121         for(int k=0; k < current_msg_size; ++k) send_buffer[k] = (char)('A' + (k %
                26));
122
123         double trial_total_latency_s = 0;
124         double trial_total_throughput_bps = 0;
125         int successful_trials_count = 0;
126
127         for (int trial = 0; trial < NUM_TRIALS; ++trial) {
128             double start_time, end_time, elapsed_time_s;
129             long successful_repetitions = 0;
130
131             printf("  Tamanho: %d Bytes, Trial %d/%d ... ", current_msg_size, trial
                    + 1, NUM_TRIALS);
132             fflush(stdout);
133
134             start_time = get_time_sec();
135             for (long rep = 0; rep < NUM_REPETITIONS; ++rep) {
136                 if (sendto(sockfd, send_buffer, current_msg_size, 0,
137                            (const struct sockaddr *)&servaddr, sizeof(servaddr)) <
                                0) {
138                     // perror("sendto failed"); // Pode poluir muito a saida
139                     continue; // Pula esta repeticao
140                 }
141                 socklen_t len = sizeof(servaddr);
142                 ssize_t n = recvfrom(sockfd, recv_buffer, RECV_BUFFER_SIZE, 0,
143                                      (struct sockaddr *)&servaddr, &len);
144                 if (n < 0) {
145                     continue; // Pula esta repeticao
146                 } else if (n != current_msg_size) {
147                     // Resposta com tamanho incorreto
148                     continue; // Pula esta repeticao
149                 }
150                 successful_repetitions++;
151             }
152             end_time = get_time_sec();
153             elapsed_time_s = end_time - start_time;
154             printf("concluido (%ld/%d reps ok)\n", successful_repetitions,
```

```
                NUM_REPETITIONS);
155
156
157             if (successful_repetitions > 0) {
158                 double current_latency_s = elapsed_time_s / successful_repetitions;
159                 double current_throughput_bps = (current_msg_size * 8.0) /
                        current_latency_s;
160                 trial_total_latency_s += current_latency_s;
161                 trial_total_throughput_bps += current_throughput_bps;
162                 successful_trials_count++;
163             } else {
164                 // printf("Trial %d para msg_size %d nao teve reflexoes bem
                        sucedidas.\n", trial + 1, current_msg_size);
165             }
166         }
167
168         if (successful_trials_count > 0) {
169             double avg_latency_s = trial_total_latency_s / successful_trials_count;
170             double avg_throughput_bps = trial_total_throughput_bps /
                    successful_trials_count;
171             printf("| %-16d | %-18.9f | %-17.2f |\n",
172                    current_msg_size, avg_latency_s, avg_throughput_bps);
173             fprintf(fp_latency_small, "%d %f\n", current_msg_size, avg_latency_s);
174             fprintf(fp_throughput_small, "%d %f\n", current_msg_size,
                    avg_throughput_bps);
175         } else {
176             printf("| %-16d | N/A                | N/A                |\n",
                    current_msg_size);
177             fprintf(fp_latency_small, "%d NaN\n", current_msg_size); //
                    Gnuplot/Python podem lidar com NaN
178             fprintf(fp_throughput_small, "%d NaN\n", current_msg_size);
179         }
180     }
181     printf("-----------------------------------------------------------\n");
182
183     // === PARTE (b): Mensagens Grandes (1KB-32KB) ===
184     printf("\nResultados para Parte (b) (UDP - Latencia e Vazao - Mensagens
            Grandes):\n");
185     printf("-------------------------------------------------------------------------\n");
186     printf("| Tam. Msg (KB) | Tam. Msg (Bytes) | Latencia Media (s) | Vazao Media
            (bps) |\n");
187     printf("-------------------------------------------------------------------------\n");
188
189     for (int i = 0; i < num_sizes_b; ++i) {
190         int current_msg_size = msg_sizes_b[i]; // Ja em bytes
191         for(int k=0; k < current_msg_size; ++k) send_buffer[k] = (char)('X' + (k %
                26));
192
193         double trial_total_latency_s = 0;
194         double trial_total_throughput_bps = 0;
195         int successful_trials_count = 0;
196
197         for (int trial = 0; trial < NUM_TRIALS; ++trial) {
198             double start_time, end_time, elapsed_time_s;
199             long successful_repetitions = 0;
200
201             printf("  Tamanho: %d KB (%d B), Trial %d/%d ... ",
                    current_msg_size/1024, current_msg_size, trial + 1, NUM_TRIALS);
202             fflush(stdout);
203
204             start_time = get_time_sec();
```

```
205             for (long rep = 0; rep < NUM_REPETITIONS; ++rep) {
206                 if (sendto(sockfd, send_buffer, current_msg_size, 0,
207                            (const struct sockaddr *)&servaddr, sizeof(servaddr)) <
                               0) {
208                     continue;
209                 }
210                 socklen_t len = sizeof(servaddr);
211                 ssize_t n = recvfrom(sockfd, recv_buffer, RECV_BUFFER_SIZE, 0,
212                                      (struct sockaddr *)&servaddr, &len);
213                 if (n < 0) {
214                     continue;
215                 } else if (n != current_msg_size) {
216                     continue;
217                 }
218                 successful_repetitions++;
219             }
220             end_time = get_time_sec();
221             elapsed_time_s = end_time - start_time;
222             printf("concluido (%ld/%d reps ok)\n", successful_repetitions,
                    NUM_REPETITIONS);
223
224             if (successful_repetitions > 0) {
225                 double current_latency_s = elapsed_time_s / successful_repetitions;
226                 double current_throughput_bps = (current_msg_size * 8.0) /
                        current_latency_s;
227                 trial_total_latency_s += current_latency_s;
228                 trial_total_throughput_bps += current_throughput_bps;
229                 successful_trials_count++;
230             } else {
231                 // printf("Trial %d para msg_size %d nao teve reflexoes bem
                        sucedidas.\n", trial + 1, current_msg_size);
232             }
233         }
234
235         if (successful_trials_count > 0) {
236             double avg_latency_s = trial_total_latency_s / successful_trials_count;
237             double avg_throughput_bps = trial_total_throughput_bps /
                    successful_trials_count;
238             printf("| %-13d | %-16d | %-18.9f | %-17.2f |\n",
239                    current_msg_size / 1024, current_msg_size, avg_latency_s,
                           avg_throughput_bps);
240             fprintf(fp_latency_large, "%d %f\n", current_msg_size, avg_latency_s);
241             fprintf(fp_throughput_large, "%d %f\n", current_msg_size,
                    avg_throughput_bps);
242         } else {
243             printf("| %-13d | %-16d | N/A                | N/A                |\n",
244                    current_msg_size / 1024, current_msg_size);
245             fprintf(fp_latency_large, "%d NaN\n", current_msg_size);
246             fprintf(fp_throughput_large, "%d NaN\n", current_msg_size);
247         }
248     }
249     printf("-----------------------------------------------------------\n");
250
251     printf("\nArquivos de dados gerados: data_latency_small.txt,
            data_throughput_small.txt, data_latency_large.txt,
            data_throughput_large.txt\n");
252
253     fclose(fp_latency_small);
254     fclose(fp_throughput_small);
255     fclose(fp_latency_large);
256     fclose(fp_throughput_large);
```

```
257     close(sockfd);
258     free(send_buffer);
259
260     return 0;
261 }
```

Listing 2: Código fonte do Testador UDP (tester_udp.c)

3.4 Testes Realizados

A execução dos testes seguiu os seguintes passos:

1. **Compilação dos Códigos:** Os programas `reflector_udp.c` e `tester_udp.c` foram compilados em seus respectivos hosts (Host B e Host A) utilizando o GCC. Comandos típicos de compilação:

   ```
   gcc reflector_udp.c -o reflector_udp -Wall
   gcc tester_udp.c -o tester_udp -Wall -lrt
   ```

   A flag `-lrt` é necessária para o `tester_udp.c` devido ao uso da função `clock_gettime`. A flag `-Wall` habilita todos os warnings comuns do compilador.

2. **Execução do Refletor (Host B):** No Host B, o programa refletor foi iniciado primeiro, para que estivesse pronto para receber conexões do testador.

   ```
   ./reflector_udp
   ```

   O refletor então começa a escutar na porta 8080, aguardando datagramas UDP.

3. **Execução do Testador (Host A):** No Host A, o programa testador foi executado, fornecendo o endereço IP do Host B como argumento de linha de comando.

   ```
   ./tester_udp <IP_DO_HOST_B>
   ```

   Por exemplo, se o IP do Host B fosse `192.168.1.101`, o comando seria:

   ```
   ./tester_udp 192.168.1.101
   ```

4. **Processo de Teste Automatizado:** O programa `tester_udp.c` automaticamente executa os testes para as duas séries de tamanhos de mensagem:

   - **Mensagens Pequenas:** Para cada tamanho no conjunto {1, 100, 200, ..., 1000} bytes.
   - **Mensagens Grandes:** Para cada tamanho no conjunto {1KB, 2KB, ..., 32KB}.

   Para cada tamanho de mensagem, o testador realiza 3 trials. Cada trial consiste em 100.000 envios de mensagem para o refletor e o aguardo da respectiva resposta. O tempo de ida e volta (RTT) para cada reflexão bem-sucedida é medido, e a latência média é calculada ao final das 100.000 repetições. A vazão é então derivada dessa latência. Os resultados dos 3 trials são então promediados para obter os valores finais de latência e vazão para aquele tamanho de mensagem.

5. **Coleta de Dados:** Durante a execução, o `tester_udp.c` imprime os resultados médios (latência e vazão) para cada tamanho de mensagem na saída padrão (esta saída foi omitida dos blocos de código LaTeX para brevidade, mas é gerada pelo programa). Simultaneamente, os dados brutos de latência e vazão para cada tamanho de mensagem testado são gravados nos seguintes arquivos:

   - `data_latency_small.txt`: Latências para mensagens pequenas.

- `data_throughput_small.txt`: Vazões para mensagens pequenas.
- `data_latency_large.txt`: Latências para mensagens grandes.
- `data_throughput_large.txt`: Vazões para mensagens grandes.

Esses arquivos são usados posteriormente para a geração dos gráficos.

3.5 Resultados Obtidos

Os resultados dos testes de latência e vazão para comunicação UDP são apresentados nas tabelas e gráficos a seguir. Os dados para as tabelas foram extraídos dos arquivos `data_latency_small.txt`, `data_throughput_small.txt`, `data_latency_large.txt` e `data_throughput_large.txt` gerados pelo programa `tester_udp.c`.

3.5.1 Tabelas de Resultados

A Tabela **??** mostra os valores médios de latência e vazão para mensagens pequenas (1 a 1000 bytes). A Tabela **??** mostra os valores médios de latência e vazão para mensagens grandes (1KB a 32KB).

Tabela 1: Resultados de Latência e Vazão UDP para Mensagens Pequenas

Tam. Msg (Bytes)	Latência Média (s)	Vazão Média (bps)
1	0.000778	17458.10
100	0.000409	1954475.62
200	0.000665	2409252.72
300	0.000856	2804292.65
400	0.000913	3505938.15
500	0.000981	4075751.89
600	0.001039	4618147.13
700	0.001100	5093252.87
800	0.001161	5510913.04
900	0.001229	5858541.58
1000	0.001321	6054038.06

Fonte: Arquivos `data_latency_small.txt` e `data_throughput_small.txt`.

Tabela 2: Resultados de Latência e Vazão UDP para Mensagens Grandes

Tam. Msg (KB)	Tam. Msg (Bytes)	Latência Média (s)	Vazão Média (bps)
1	1024	0.001336	6130859.94
2	2048	0.001360	12047771.52
3	3072	0.001519	16183290.46
4	4096	0.001650	19864838.92
5	5120	0.001220	35532021.06
6	6144	0.001021	48129913.40
7	7168	0.001080	53088947.45
8	8192	0.001138	57579835.82
9	9216	0.001245	59290060.90
10	10240	0.001408	58164694.93
11	11264	0.001441	62519381.24
12	12288	0.001533	64125773.32
13	13312	0.001593	66856133.76
14	14336	0.001773	64667732.10
15	15360	0.001837	66887399.05
16	16384	0.001904	68844202.57
17	17408	0.001973	70587189.37
18	18432	0.002101	70181242.90
19	19456	0.002175	71557447.41
20	20480	0.002217	73899730.32
21	21504	0.002325	74007672.54
22	22528	0.002383	75619850.29
23	23552	0.002528	74560954.97
24	24576	0.002598	75678326.17
25	25600	0.002674	76577456.95
26	26624	0.002796	76189986.26
27	27648	0.002866	77190103.20
28	28672	0.002917	78636641.49
29	29696	0.003005	79046213.57
30	30720	0.003068	80108435.17
31	31744	0.003196	79451981.93
32	32768	0.003253	80578406.75

Fonte: Arquivos `data_latency_large.txt` e `data_throughput_large.txt`.

3.5.2 Gráficos de Resultados

Os dados apresentados nas tabelas foram utilizados para gerar os seguintes gráficos. Os gráficos devem ser gerados utilizando os arquivos de dados e inseridos aqui.

graficos/latencia_pequenas.png

Figura 1: Latência UDP em função do Tamanho da Mensagem (Pequenas)

graficos/vazao_pequenas.png

Figura 2: Vazão UDP em função do Tamanho da Mensagem (Pequenas)

graficos/latencia_grandes.png

Figura 3: Latência UDP em função do Tamanho da Mensagem (Grandes)

Figura 4: Vazão UDP em função do Tamanho da Mensagem (Grandes)

3.6 Análise dos Resultados Obtidos e Conclusões

A análise dos resultados obtidos (tabelas e gráficos) permite observar o comportamento da latência e da vazão na comunicação UDP em função do tamanho da mensagem.

Latência:

- **Mensagens Pequenas (Figura ?? e Tabela ??):** Observa-se que a latência para a mensagem de 1 byte (0.000778 s) é notavelmente maior que para 100 bytes (0.000409 s). Este comportamento pode ser atribuído a overheads fixos de processamento de pacotes que são mais significativos para payloads mínimos, ou a otimizações no sistema operacional/pilha de rede que lidam de forma diferente com pacotes extremamente pequenos versus pacotes um pouco maiores. A partir de 100 bytes até 1000 bytes, a latência mostra uma tendência geral de aumento (de 0.000409 s para 0.001321 s), o que é esperado, pois mais dados precisam ser processados, serializados e transmitidos.
- **Mensagens Grandes (Figura ?? e Tabela ??):** Para mensagens grandes, a latência geralmente aumenta com o tamanho da mensagem, como visto na passagem de 1KB (0.001336 s) para 32KB (0.003253 s). No entanto, a progressão não é estritamente monotônica. Por exemplo, há uma queda notável na latência para mensagens de 5KB (0.001220 s) e 6KB (0.001021 s) em comparação com 4KB (0.001650 s).

Essas variações podem ser devidas a múltiplos fatores, incluindo a interação com os buffers de rede (tanto no transmissor quanto no receptor), o agendamento de pacotes pelo sistema operacional, e possivelmente a dinâmica da rede Wi-Fi, que pode ter variações de desempenho. O tempo de transmissão do payload torna-se um componente mais dominante da latência total para mensagens maiores.

Vazão (Throughput):

- **Mensagens Pequenas (Figura ?? e Tabela ??):** A vazão aumenta acentuadamente com o tamanho da mensagem nesta faixa. Para 1 byte, a vazão é de aproximadamente 17.46 Kbps, enquanto para 1000 bytes, atinge cerca de 6.05 Mbps. Este aumento é esperado, pois o overhead fixo por pacote (cabeçalhos IP/UDP, tempo de processamento) é amortizado por uma quantidade maior de dados úteis à medida que o tamanho da mensagem cresce.
- **Mensagens Grandes (Figura ?? e Tabela ??):** A vazão continua a aumentar com o tamanho da mensagem, passando de aproximadamente 6.13 Mbps para 1KB até cerca de 80.58 Mbps para 32KB. A taxa de aumento da vazão parece diminuir ligeiramente para os tamanhos de mensagem maiores, sugerindo que pode estar se aproximando de um limite imposto pela capacidade da rede local, pela velocidade de processamento dos hosts, ou pelas características do protocolo UDP. As flutuações observadas na latência (e.g., em 5KB e 6KB) refletem-se diretamente na vazão, causando picos correspondentes (e.g., 35.53 Mbps para 5KB e 48.13 Mbps para 6KB).

Considerações Gerais:

- **Relação Latência e Vazão:** A vazão é calculada como (Tamanho da Mensagem * 8) / Latência. Portanto, as variações na latência têm um impacto inverso e direto na vazão calculada.
- **Protocolo UDP:** Sendo UDP um protocolo "best-effort", sem garantias de entrega ou controle de congestionamento, as medições podem ser influenciadas por perdas de pacotes não detectadas pelo simples eco (embora o programa conte reflexões bem-sucedidas). Em uma rede local com baixa taxa de erro, este efeito tende a ser minimizado.
- **Overheads:** Os cabeçalhos UDP (8 bytes) e IPv4 (20 bytes, sem opções) somam 28 bytes de overhead por datagrama. Para a mensagem de 1 byte, o payload representa apenas $1/(1+28) \approx 3.4\%$ do total de bytes transmitidos na camada de rede, explicando a baixa vazão. Para uma mensagem de 1000 bytes, o payload é $1000/(1000+28) \approx 97.3\%$, resultando em maior eficiência.
- **Limitações do Ambiente e Medição:** Os testes foram realizados em uma rede WLAN, que é suscetível a interferências e variações de desempenho. A carga nos hosts A e B, embora se espere que seja mínima durante os testes dedicados, e as características específicas do hardware de rede (roteador, adaptadores Wi-Fi) influenciam os resultados. O uso de `clock_gettime(CLOCK_MONOTONIC, ...)` é crucial para medições de tempo precisas.
- **Anomalias e Justificativas:** A latência menor para 100 bytes em comparação com 1 byte, e as quedas de latência (com consequentes picos de vazão) para mensagens de 5KB e 6KB são os pontos mais notáveis. As causas exatas podem envolver otimizações de baixo nível na pilha de rede, alinhamento de buffers, ou interações com o scheduler do sistema operacional que favorecem certos tamanhos de pacotes ou fluxos de dados. Sem ferramentas de profiling mais profundas do kernel ou da rede, é difícil determinar a causa exata dessas não monotonicidades.

Conclusões Finais: O experimento demonstrou com sucesso a relação entre tamanho da mensagem, latência e vazão para o protocolo UDP em uma rede local. Conforme esperado, a vazão tende a aumentar com o tamanho da mensagem, pois os overheads fixos são melhor amortizados. A latência também tende a aumentar com o tamanho da mensagem, especialmente para payloads maiores, devido ao tempo de transmissão. No entanto, o comportamento não é sempre linear ou monotônico, indicando a complexidade das interações dentro dos hosts e na rede. Os resultados ressaltam que, para aplicações UDP, a escolha do tamanho da mensagem pode ter um impacto significativo no desempenho percebido, e que as características específicas do ambiente de rede e dos sistemas envolvidos podem levar a comportamentos não intuitivos que merecem consideração no design de aplicações.

Referências

[1] TANENBAUM, Andrew S.; WETHERALL, David J. *Redes de Computadores*. 5. ed. São Paulo: Pearson, 2011. ISBN 9788576059980.

[2] POSTEL, Jon. *Internet Control Message Protocol*. RFC 792. 1981. Disponível em: `https://www.rfc-editor.org/rfc/rfc792.txt`. Acesso em: 12 maio 2025.

[3] BEVERLY, Robert. *Yarrp'ing the Internet: Randomized High-Speed Active Topology Discovery*. 2016. Disponível em: `https://arxiv.org/abs/1605.03999`. Acesso em: 12 maio 2025.

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

[5] STEVENS, W. Richard; FENNER, Bill; RUDOFF, Andrew M. *UNIX Network Programming, Volume 1: The Sockets Networking API*. 3. ed. Addison-Wesley Professional, 2003.

[6] Linux Programmer's Manual. *man pages* para `socket(2)`, `sendto(2)`, `recvfrom(2)`, `bind(2)`, `setsockopt(2)`, `clock_gettime(2)`, `inet_pton(3)`. Disponível online ou via comando `man` no sistema Linux.