Centro Federal de Educação Tecnológica de Minas Gerais
Departamento de Engenharia de Computação
Engenharia de Computação

Aproveitamento do curso de Princípios de Comunicação de Dados

Leonardo de Oliveira Campos

Divinópolis

Agosto/2024

Sumário

1 Introdução do Trabalho

Neste trabalho de aproveitamento de estudos de Princípios de Comunicação de Dados, buscou-se aprofundar no Modelo Internet (TCP/IP), o qual parte do princípio de 5 camadas de abstração, no qual é possível descrever todo o seu funcionamento. Diferente do modelo de Interconexão de Sistemas Abertos (OSI), do inglês **Over The Air**, o qual tem 7 camadas de abstração.

O Modelo OSI é um modelo teórico que fornece uma estrutura conceitual para entender e projetar redes de comunicação, dividindo as funções de comunicação em sete camadas distintas. Por outro lado, o Modelo Internet, comumente conhecido como TCP/IP, é o modelo prático atualmente utilizado para as comunicações de dados na Internet. Este modelo é composto por quatro camadas e é a base da arquitetura de rede da Internet, definindo protocolos e padrões que permitem a comunicação entre dispositivos em uma rede global (Forouzan e Fegan 2010).

Antes de vermos as diferenças entre os modelos, para melhor entendimento da matéria, é importante frizar que as informações para confecção desse material de estudo foram baseadas em 5 livros de Redes, sendo eles: (Forouzan e Fegan 2010) (Tanenbaum 2003) (Kurose e Ross 2013) (Maia 2013) (Peterson e Davie 2013).

1.1 OSI

Estabelecida em 1947, a International Organization for Standardization (ISO) é uma entidade dedicada à definição de normas internacionais globais, com a participação de diversas nações. Um dos padrões ISO mais relevantes para as comunicações de dados em redes é o modelo OSI (Open Systems Interconnection), introduzido no final da década de 1970 (Forouzan e Fegan 2010).

O modelo OSI define um sistema aberto como um conjunto de protocolos que permite a comunicação entre dois sistemas distintos, independentemente de suas

arquiteturas subjacentes. O objetivo do modelo OSI é facilitar a interoperabilidade entre diferentes sistemas sem exigir modificações na lógica do hardware e software de cada um. Vale ressaltar que o modelo OSI não é um protocolo específico, mas sim uma estrutura conceitual destinada a compreender e projetar arquiteturas de rede que sejam flexíveis, robustas e interoperáveis (Forouzan e Fegan 2010).

O modelo OSI é uma estrutura em camadas projetada para a criação de sistemas de redes que possibilitam a comunicação entre diversos tipos de sistemas de computadores. Ele é composto por sete camadas distintas, mas inter-relacionadas, cada uma das quais define um aspecto específico do processo de transferência de informações através de uma rede, como pode ser visto na **??**. Entender os fundamentos do modelo OSI oferece uma base sólida para a exploração de outros conceitos relacionados às comunicações de dados.

Figura 1 – Modelo OSI. Fonte: (Forouzan e Fegan 2010).

O modelo OSI é composto por sete camadas ordenadas: física (camada 1), enlace (camada 2), rede (camada 3), transporte (camada 4), sessão (camada 5), apresentação (camada 6) e aplicação (camada 7). A **??** ilustra as camadas envolvidas quando uma mensagem é enviada do dispositivo A para o dispositivo B. Durante a

transmissão, a mensagem pode passar por vários nós intermediários, que geralmente operam apenas nas três primeiras camadas do modelo OSI.

Os projetistas do modelo OSI analisaram o processo de transmissão de dados em seus elementos mais fundamentais. Eles identificaram funções de rede com propósitos relacionados e organizaram essas funções em grupos distintos, formando as camadas do modelo. Cada camada define um conjunto de funções específicas, distintas das demais camadas. Esse design permite que o modelo OSI seja tanto abrangente quanto flexível, facilitando a interoperabilidade entre sistemas anteriormente incompatíveis.

Dentro de uma máquina, cada camada utiliza os serviços da camada imediatamente inferior e fornece serviços à camada superior. Por exemplo, a camada 3 utiliza os serviços da camada 2 e oferece serviços à camada 4. Entre máquinas, a camada x em uma máquina comunica-se com a camada x da outra máquina. Essa comunicação é regida por um conjunto de regras e convenções conhecidas como protocolos. Os processos em cada máquina que interagem em uma camada específica são chamados de processos peer-to-peer. Assim, a comunicação entre máquinas é um processo peer-to-peer que utiliza os protocolos apropriados para a camada em questão.

1.1.1 *peer-to-peer*

No modelo OSI, *peer-to-peer* refere-se à comunicação entre processos em máquinas diferentes, onde cada processo opera na mesma camada do modelo OSI e utiliza protocolos específicos dessa camada para trocar informações. Em uma comunicação *peer-to-peer*, cada máquina age como um ”igual”em relação às outras, e a comunicação entre elas segue as regras e convenções estabelecidas pelos protocolos da camada correspondente.

1.1.2 Camadas do Modelo OSI

1. **Camada Física (Camada 1):** Trata da transmissão de bits brutos através do meio físico. Define aspectos elétricos e mecânicos da transmissão, como volta-

gem e tipo de cabos.

2. **Camada de Enlace (Camada 2):** Responsável pela transmissão de quadros (frames) entre dois dispositivos conectados diretamente. Inclui o controle de erros e a identificação de endereços de hardware.

3. **Camada de Rede (Camada 3):** Gerencia a entrega de pacotes entre dispositivos em diferentes redes. Inclui o roteamento e a escolha do melhor caminho para a transmissão dos dados.

4. **Camada de Transporte (Camada 4):** Garante a entrega correta e ordenada dos pacotes entre dois sistemas finais. Inclui o controle de fluxo e a correção de erros.

5. **Camada de Sessão (Camada 5):** Estabelece, gerencia e encerra sessões de comunicação entre aplicativos. Garante que os dados sejam sincronizados e coordenados.

6. **Camada de Apresentação (Camada 6):** Traduz e formata dados para que possam ser compreendidos pelas camadas superiores. Inclui a criptografia e a compressão de dados.

7. **Camada de Aplicação (Camada 7):** Fornece serviços diretamente aos aplicativos de rede, como e-mail e navegação web. É a camada mais próxima do usuário final.

1.2 TCP/IP

A ARPANET, predecessora da Internet, foi uma rede de pesquisa criada pelo Departamento de Defesa dos Estados Unidos (DoD). Inicialmente conectando universidades e instituições por linhas telefônicas, a ARPANET enfrentou desafios com protocolos existentes conforme surgiam novas tecnologias, como redes de rádio e satélite (Tanenbaum 2003). Para resolver esses problemas, foi desenvolvido o Modelo de Referência TCP/IP, introduzido em 1974 (Tanenbaum 2003).

O TCP/IP é composto por quatro camadas: host-rede, internet, transporte e aplicação. Em comparação com o modelo OSI, a camada host-rede corresponde às camadas física e de enlace de dados, a camada internet equivale à camada de rede, e a camada de aplicação cobre as funções das camadas de sessão, apresentação e aplicação do OSI. A camada de transporte do TCP/IP também lida com parte das funções da camada de sessão. Portanto, o TCP/IP pode ser simplificadamente descrito com cinco camadas: física, enlace, rede, transporte e aplicação, com a camada de aplicação englobando as três camadas superiores do modelo OSI (Forouzan e Fegan 2010).

1.2.1 Camadas do Modelo TCP/IP

- **Camada de Host-Rede:**
  - Equivalente às camadas física e de enlace de dados do modelo OSI.
  - Responsável pela transmissão de bits através do meio físico e pela comunicação de dados entre dispositivos diretamente conectados.
  - Inclui protocolos como Ethernet e PPP (Point-to-Point Protocol).
- **Camada Internet:**
  - Equivalente à camada de rede do modelo OSI.
  - Gerencia o roteamento e o endereçamento de pacotes entre redes diferentes.

  - Inclui o protocolo IP (Internet Protocol), que é fundamental para a comunicação entre sistemas em redes distintas.

- **Camada de Transporte:**
  - Equivalente à camada de transporte do modelo OSI, mas também lida com algumas funções da camada de sessão.
  - Garante a entrega correta e ordenada dos pacotes entre sistemas finais.
  - Inclui os protocolos TCP (Transmission Control Protocol) e UDP (User Datagram Protocol).

- **Camada de Aplicação:**
  - Cobre as funções das camadas de sessão, apresentação e aplicação do modelo OSI.
  - Fornece serviços diretamente aos aplicativos de rede, como e-mail, transferência de arquivos e navegação web.
  - Inclui protocolos como HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol) e SMTP (Simple Mail Transfer Protocol).

1.3 Quais as diferenças entre os dois modelos

- **Número de Camadas:**
  - **Modelo OSI:** Possui 7 camadas (física, enlace, rede, transporte, sessão, apresentação e aplicação).
  - **Modelo TCP/IP:** Originalmente possui 4 camadas (host-rede, internet, transporte e aplicação). Algumas fontes podem descrevê-lo com 5 camadas, separando a camada física do enlace.
- **Correspondência das Camadas:**
  - **Modelo OSI:** Cada camada é distinta e tem funções específicas. As camadas superiores e inferiores são claramente definidas.
  - **Modelo TCP/IP:** As camadas podem combinar funções de várias camadas do OSI. Por exemplo, a camada de aplicação do TCP/IP cobre funções das camadas de aplicação, apresentação e sessão do OSI.
- **Objetivo e Histórico:**
  - **Modelo OSI:** Desenvolvido para fornecer uma abordagem teórica e padronizada para redes de computadores, independente de tecnologia específica.
  - **Modelo TCP/IP:** Desenvolvido para resolver problemas práticos de comunicação entre redes e facilitar a interoperabilidade entre diferentes sistemas de rede.
- **Desenvolvimento e Uso:**
  - **Modelo OSI:** Desenvolvido pela ISO e utilizado principalmente como um modelo de referência para compreensão e padronização de redes.
  - **Modelo TCP/IP:** Desenvolvido pelo Departamento de Defesa dos EUA e amplamente adotado na prática como a base da Internet e de muitas redes modernas.

- **Abordagem:**
  - **Modelo OSI:** Mais teórico e detalhado, com foco em descrever como as redes devem ser estruturadas.
  - **Modelo TCP/IP:** Mais pragmático, com foco em como os protocolos devem funcionar na prática para garantir a comunicação eficiente e confiável.
- **Protocólos:**
  - **Modelo OSI:** Define protocolos e funções de forma mais geral e teórica.
  - **Modelo TCP/IP:** Inclui protocolos específicos e implementações reais, como IP, TCP, UDP e HTTP.

2 Ferramentas usadas

2.1 MATLAB Online

O MATLAB Online é uma versão baseada em navegador da popular plataforma de computação numérica MATLAB. Permite que usuários acessem e utilizem as funcionalidades do MATLAB diretamente de um navegador web, sem a necessidade de instalação local. É particularmente útil para análise de dados, desenvolvimento de algoritmos e criação de modelos matemáticos, proporcionando um ambiente de trabalho acessível e colaborativo.

2.2 Docker

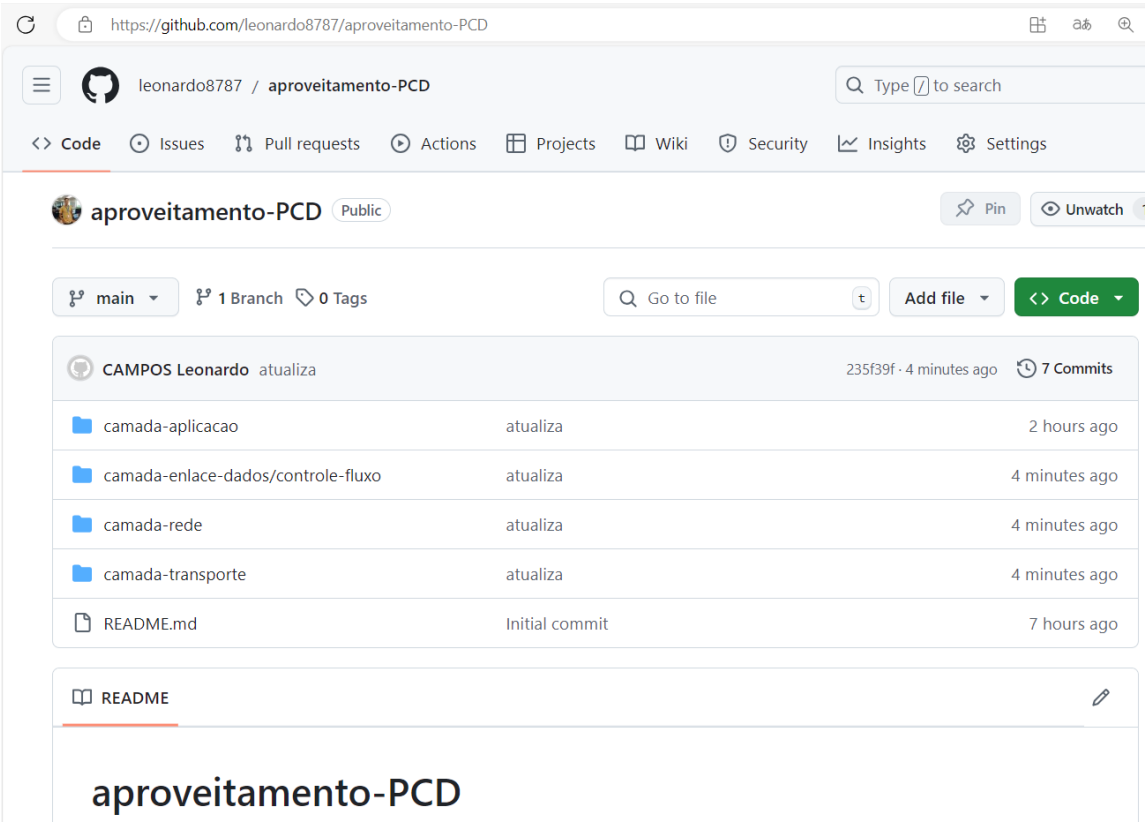
Docker é uma plataforma de software que permite a criação, execução e gerenciamento de contêineres. Contêineres são unidades leves e portáteis que encapsulam uma aplicação e suas dependências, garantindo que funcionem de maneira consistente em diferentes ambientes.

2.3 Wireshark

Wireshark é uma ferramenta de análise de rede que permite capturar e inspecionar pacotes de dados que trafegam por uma rede. Utilizada para diagnóstico e resolução de problemas de rede, Wireshark fornece uma visão detalhada do tráfego de rede, ajudando na identificação de problemas, na análise de protocolos e na segurança da rede.

3 Repositório com as aplicações

Todas as aplicações feitas neste trabalho de aproveitamento, estão disponibilizadas no seguinte link: <https://github.com/leonardo8787/aproveitamento-PCD>. Neste repositório é possível fazer a instalação do docker para rodar todas as aplicações em python com as bibliotecas já instaladas.



4 Camada Física

Responsável pela transmissão de dados brutos através do meio físico, como cabos de cobre, fibras ópticas ou sinais sem fio. Ela trata da codificação dos dados em sinais, a taxa de transmissão, e a topologia física da rede (Forouzan e Fegan 2010).

Nessa seção foi escolhido trabalhar com o MATLAB e a visualização das fases de sinal. Vale ressaltar que o professor não adicionou os experimentos nas relações passadas na introdução deste trabalho de aproveitamento de curso, mas ressaltou a liberdade do aluno em criar e/ou utilizar de exemplos dispostos no material de referência. Para tal, irá ser tratado os temas: Modulação, Propagação e Multiplexação.

material: MATLAB e Slides da Universidade Federal do Rio Grande do Norte, disponível em: <https://encurtador.com.br/JBvPp>.

4.1 Modulação de Sinais usando MATLAB/Simulink

A modulação de sinais refere-se à técnica pela qual as características de uma onda portadora são alteradas com o objetivo de transmitir informações. Essa modificação pode envolver a amplitude, frequência ou fase da onda, dependendo do tipo de modulação empregada. O uso adequado de técnicas de modulação permite uma comunicação mais eficiente, seja em sistemas analógicos ou digitais (Peterson e Davie 2013).

4.1.1 Tipos de Modulação

Existem diferentes técnicas de modulação, cada uma com características específicas que a tornam adequada para determinadas aplicações:

4.1.2 Modulação de Amplitude (AM)

Na modulação de amplitude, a informação é transmitida variando-se a amplitude da onda portadora em função do sinal de entrada. Esse tipo de modulação é amplamente utilizado em sistemas de radiodifusão, como em emissoras de rádio AM. Embora seja uma técnica relativamente simples, a modulação AM é mais suscetível a ruídos e interferências.

4.1.3 Modulação de Frequência (FM)

A modulação de frequência consiste em alterar a frequência da onda portadora de acordo com o sinal de informação. Esse método é bastante utilizado em transmissões de rádio FM, onde a resistência ao ruído é um fator decisivo. Devido à sua robustez, a modulação FM é preferida em muitas aplicações onde a qualidade do sinal é crítica.

4.1.4 Modulação de Fase (PM)

Na modulação de fase, a fase da onda portadora é modificada conforme o sinal que se deseja transmitir. Embora menos comum em aplicações isoladas, a modulação de fase é fundamental em esquemas de modulação mais complexos, como a modulação em quadratura, usada em sistemas de comunicação digital.

4.1.5 Modulação por Pulso (PWM, PAM)

A modulação por pulso envolve a variação de parâmetros de pulsos de sinal, como largura ou amplitude, para transmitir informações. Esse tipo de modulação é amplamente utilizado em sistemas de controle e em comunicações digitais, onde a integridade do sinal e a eficiência de transmissão são cruciais.

4.1.6 Importância e Aplicações

A modulação de sinais é essencial para a transmissão de informações a longas distâncias, tornando os sinais adequados para diferentes meios de transmissão, como cabos, fibras ópticas ou ondas de rádio. Além disso, a modulação permite a multiplexação, onde múltiplos sinais são transmitidos simultaneamente por uma única portadora, otimizando a largura de banda disponível e minimizando interferências.

Entre as aplicações mais notáveis da modulação, destacam-se:

- **Rádio e TV**: Uso de modulação AM e FM para transmissão de áudio e vídeo.
- **Telecomunicações**: Modulação digital (como PSK e QAM) em sistemas modernos de comunicação, incluindo telefonia móvel e internet.
- **Controle e Automação**: Modulação por largura de pulso (PWM) em controle de motores e fontes de alimentação.

4.1.7 Implementação em MATLAB

Neste tópico, é apresentado uma explicação detalhada do código de modulação de sinais em MATLAB. O código implementa a modulação de um sinal usando uma portadora e realiza a plotagem dos sinais modulante, portadora e modulado.

4.1.8 Frequência e Período de Amostragem

```
% Frequência de amostragem do sinal
Fs = 40000;

% Período de amostragem do sinal
Ts = 1 / Fs;
```

- **Fs**: Define a frequência de amostragem como 40.000 amostras por segundo. A frequência de amostragem é a quantidade de amostras por segundo utilizadas para digitalizar um sinal contínuo.
- **Ts**: É o período de amostragem, que é o inverso da frequência de amostragem. Representa o intervalo de tempo entre duas amostras consecutivas.

4.1.9 Variável Tempo

```
% Variável tempo amostrada
t = 0:Ts:2-Ts;
```

- **t**: Cria um vetor de tempo que vai de 0 a (2-Ts) segundos, com incrementos de Ts. Isso significa que o vetor t tem amostras ao longo de 2 segundos.

4.1.10 Sinal Modulante e Portadora

```
% Sinal modulante de frequência angular de 1000 rad/s
m = cos(1000 * t);

% Sinal da portadora
p = cos(10000 * t);
```

- **m**: Define o sinal modulante como uma onda cossenoidal com frequência angular de 1000 rad/s. Este é o sinal que contém as informações que você deseja transmitir.
- **p**: Define o sinal da portadora como uma onda cossenoidal com frequência angular de 10000 rad/s. A portadora é usada para transportar o sinal modulante através do meio de transmissão.

4.1.11 Sinal Modulado

```
% Sinal modulado
x = m .* p;
```

- **x**: O sinal modulado é obtido multiplicando o sinal modulante m pelo sinal da portadora p. Isso gera um novo sinal x que contém a informação do sinal modulante, mas que está "embutida"na frequência da portadora.

4.1.12 Plotagem dos Gráficos com Zoom

```
figure;

subplot(3,1,1);
plot(t, m, 'b', 'LineWidth', 1);
xlabel('t');
ylabel('m(t)');
title('Sinal Modulante');
xlim([0 0.01]); % Zoom no eixo x
ylim([-1.5 1.5]); % Ajuste no eixo y
```

- **Primeiro gráfico** (subplot 1): Mostra o sinal modulante $m(t)$, que é uma onda cossenoidal de baixa frequência. Esse sinal representa a informação que você quer transmitir 2.

```
subplot(3,1,2);
plot(t, p, 'y', 'LineWidth', 1);
xlabel('t');
ylabel('p(t)');
title('Sinal da Portadora');
xlim([0 0.01]); % Zoom no eixo x
ylim([-1.5 1.5]); % Ajuste no eixo y
```

- **Segundo gráfico** (subplot 2): Mostra o sinal da portadora $p(t)$, que é uma onda cossenoidal de alta frequência. Esse sinal é usado para transportar o sinal modulante através do meio de transmissão (por exemplo, em uma transmissão de rádio) 2.

```
subplot(3,1,3);
plot(t, x, 'g', 'LineWidth', 1);
xlabel('t');
ylabel('x(t)');
title('Sinal Modulado');
xlim([0 0.01]); % Zoom no eixo x
ylim([-1.5 1.5]); % Ajuste no eixo y
```

- **Terceiro gráfico** (subplot 3): Mostra o sinal modulado $x(t)$, que é o resultado da multiplicação do sinal modulante pela portadora. Neste sinal, você pode observar que ele oscila na frequência da portadora, mas com amplitude variando de acordo com o sinal modulante 2.

4.1.13 Resumo Visual

- **Sinal Modulante** $m(t)$: Uma onda de baixa frequência representando a informação original.
- **Sinal da Portadora** $p(t)$: Uma onda de alta frequência usada para transportar o sinal modulante.
- **Sinal Modulado** $x(t)$: A combinação do sinal modulante e da portadora, onde a informação original está "embutida"na onda de alta frequência.

Os ajustes de zoom nos eixos x e y permitem que você visualize melhor as oscilações e a interação entre o sinal modulante e a portadora no sinal modulado 2.

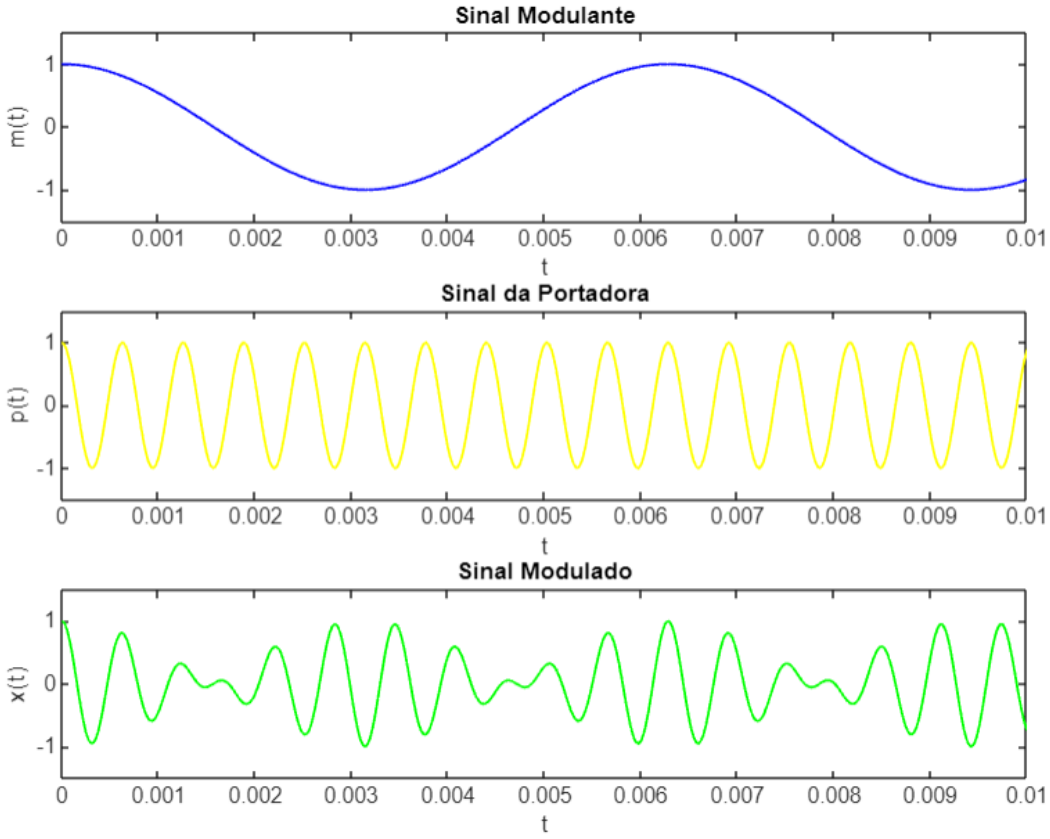


Figura 2 – Fases da modulação. Fonte: Feito pelo próprio autor.

4.2 Simulação de propagação de sinais em MATLAB/Simulink

4.2.1 O que é Propagação de Sinais

A **propagação de sinais** refere-se ao processo pelo qual um sinal, que pode ser uma onda elétrica, eletromagnética ou acústica, se move através de um meio, como o espaço livre, cabos, fibras ópticas, ou outros canais de comunicação. Durante a propagação, o sinal pode ser afetado por diversos fatores, incluindo atenuação, ruído, e distorção (Peterson e Davie 2013).

- **Atenuação**: É a redução da amplitude do sinal à medida que ele viaja através do meio. Pode ser causada por absorção, dispersão ou perda de energia.
- **Ruído**: São sinais indesejados que se misturam com o sinal desejado, prejudicando a qualidade e a clareza do sinal recebido.
- **Distorção**: Alterações na forma do sinal original durante a propagação, que podem ocorrer devido a variações no meio ou no caminho do sinal.

4.2.2 Como a Propagação é Feita

Para simular a propagação de sinais, um modelo matemático é usado para replicar como um sinal se comporta em um canal de comunicação. A simulação geralmente envolve os seguintes passos:

1. **Geração do Sinal de Portadora**: Um sinal de alta frequência, conhecido como portadora, é gerado. Este sinal transporta a informação através do canal. 2. **Aplicação de Atenuação**: O sinal é atenuado para simular a perda de amplitude que ocorre durante a propagação no meio. 3. **Adição de Ruído**: Ruído é adicionado ao sinal para simular a interferência que ocorre em canais reais.

Esses passos podem ser vistos na 3, para melhor entendimento.

4.2.3 Explicação do Código MATLAB

O código MATLAB a seguir simula a propagação de um sinal e visualiza os efeitos da atenuação e do ruído:

```
% Configura  es do sinal
Fs = 1e6;          % Frequ ncia de amostragem (1 MHz)
T = 1e-3;          % Dura   o do sinal (1 ms)
t = 0:1/Fs:T-1/Fs; % Vetor de tempo

% Sinal da portadora (exemplo: senoide de 100 kHz)
fc = 100e3;        % Frequ ncia da portadora (100 kHz)
sinal_tx = cos(2*pi*fc*t); % Sinal transmitido (portadora)

% Simula   o da propaga   o com atenua   o
atenuacao = 0.5;   % Fator de atenua   o (0.5 para reduzir a
   amplitude do sinal)
sinal_rx = atenuacao * sinal_tx; % Sinal recebido ap s
   atenua   o

% Adicionando ru do branco Gaussiano ao sinal manualmente
SNR = 20; % Rela   o sinal-ru do em dB
potencia_sinal = mean(sinal_rx.^2); % Pot ncia do sinal
potencia_ruido = potencia_sinal / (10^(SNR/10)); % Pot ncia do
    ru do
ruido = sqrt(potencia_ruido) * randn(size(sinal_rx)); % Gera
   ru do
sinal_rx_ruidoso = sinal_rx + ruido; % Adiciona ru do ao sinal

% Plotagem dos sinais
figure;

% Definindo o intervalo para o zoom
zoom_start = 0;
zoom_end = 0.0001; % Exibindo apenas os primeiros 0.1 ms

subplot(3,1,1);
plot(t, sinal_tx, 'b');
```

```
30 xlabel('Tempo (s)');
31 ylabel('Amplitude');
32 title('Sinal Transmitido (Portadora)');
33 xlim([zoom_start zoom_end]); % Aplicando zoom no eixo x
34
35 subplot(3,1,2);
36 plot(t, sinal_rx, 'g');
37 xlabel('Tempo (s)');
38 ylabel('Amplitude');
39 title('Sinal Recebido ap s Atenua  o');
40 xlim([zoom_start zoom_end]); % Aplicando zoom no eixo x
41
42 subplot(3,1,3);
43 plot(t, sinal_rx_ruidoso, 'r');
44 xlabel('Tempo (s)');
45 ylabel('Amplitude');
46 title('Sinal Recebido com Ru do');
47 xlim([zoom_start zoom_end]); % Aplicando zoom no eixo x
```

4.2.4 Configuração do Sinal

O código começa com a configuração do sinal:

- `Fs = 1e6;` define a frequência de amostragem como 1 MHz, o que significa que o sinal é amostrado 1 milhão de vezes por segundo.
- `T = 1e-3;` define a duração do sinal como 1 ms. Portanto, o sinal é observado por 1 milissegundo.
- `t = 0:1/Fs:T-1/Fs;` cria um vetor de tempo que vai de 0 até `T-1/Fs` com incrementos de `1/Fs`, definindo o intervalo entre as amostras.

4.2.5 Sinal da Portadora

A seção seguinte gera o sinal da portadora:

- `fc = 100e3;` define a frequência da portadora como 100 kHz.
- $sinal_tx = cos(2 * pi * fc * t)$; gera o sinal da portadora utilizando uma função cosseno com frequência de 100 kHz.

4.2.6 Simulação da Propagação com Atenuação

Aqui, o código simula a atenuação do sinal:

- `atenuacao = 0.5;` define um fator de atenuação de 0.5, reduzindo a amplitude do sinal pela metade.
- $sinal_rx = atenuacao * sinal_tx$; aplica a atenuação ao sinal transmitido.

4.2.7 Adição de Ruído Branco Gaussiano

Esta parte do código adiciona ruído ao sinal:

- `SNR = 20;` define a relação sinal-ruído em 20 dB.
- $potencia_sinal = mean(sinal_rx.^2)$; calcula a potência do sinal recebido.
- $potencia_ruido = potencia_sinal/(10^(SNR/10))$; determina a potência do ruído com base na relação sinal-ruído.
- $ruido = sqrt(potencia_ruido) * randn(size(sinal_rx))$; gera o ruído branco Gaussiano com a potência calculada.
- $sinal_rx_ruidoso = sinal_rx + ruido$; adiciona o ruído ao sinal recebido.

4.2.8 Plotagem dos Sinais

Finalmente, o código plota os sinais:

- `figure;` cria uma nova figura para a plotagem.

- `subplot(3,1,1);` seleciona a primeira área da subplot para o sinal transmitido.
- $plot(t, sinal_tx, 'b')$; plota o sinal transmitido em azul.
- $xlim([zoom_startzoom_end])$; aplica zoom ao intervalo definido para a visualização.
- *subplot*(3,1,2); seleciona a segunda área da subplot para o sinal recebido após atenuação.
- $plot(t, sinal_rx, 'g')$; plota o sinal recebido em verde.
- *subplot*(3,1,3); seleciona a terceira área da subplot para o sinal recebido com ruído.
- $plot(t, sinal_rx_ruidoso, 'r')$; plota o sinal recebido com ruído em vermelho.

A explicação pode ser melhor entendida observando a 3.

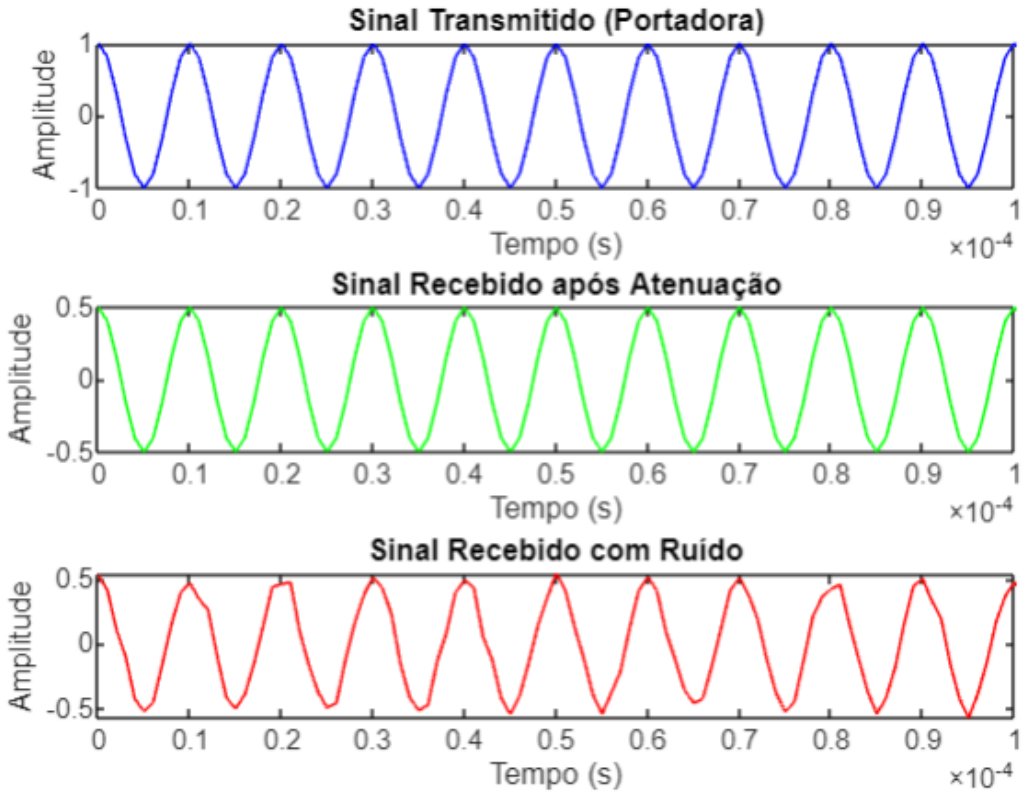


Figura 3 – Fases da propagação. Fonte: Feito pelo próprio autor.

4.3 Multiplexação de Sinais

A multiplexação é uma técnica utilizada em sistemas de comunicação para combinar múltiplos sinais em um único canal de transmissão. Isso permite que vários sinais sejam enviados simultaneamente através de um único meio de comunicação, otimizando o uso da largura de banda e aumentando a eficiência do sistema. Existem diferentes tipos de multiplexação, como Multiplexação por Divisão de Tempo (TDM), Multiplexação por Divisão de Frequência (FDM), e Multiplexação por Divisão de Código (CDM) (Maia 2013).

Neste documento, vamos focar na Multiplexação por Divisão de Tempo (TDM), que é uma técnica onde o tempo é dividido em intervalos discretos, e cada sinal é transmitido em intervalos específicos.

4.3.1 Explicação do Código MATLAB

O código MATLAB a seguir demonstra um exemplo de multiplexação TDM, onde três sinais distintos são multiplexados e depois demultiplexados.

```
% Configura  es do sinal
Fs = 1e4;          % Frequ ncia de amostragem (10 kHz)
T = 1;             % Dura   o do sinal (1 s)
t = 0:1/Fs:T-1/Fs; % Vetor de tempo
```

4.3.2 Configurações do Sinal

- `Fs = 1e4;` define a frequência de amostragem como 10 kHz, indicando que o sinal é amostrado 10 mil vezes por segundo.
- `T = 1;` define a duração do sinal como 1 segundo.
- `t = 0:1/Fs:T-1/Fs;` cria um vetor de tempo que vai de 0 até `T-1/Fs` com incrementos de `1/Fs`, correspondendo ao intervalo de tempo de 1 segundo amostrado a 10 kHz.

```
% Sinais individuais
f1 = 1;            % Frequ ncia do sinal 1 (1 Hz)
f2 = 3;            % Frequ ncia do sinal 2 (3 Hz)
f3 = 5;            % Frequ ncia do sinal 3 (5 Hz)

sinal1 = sin(2*pi*f1*t); % Sinal 1
sinal2 = sin(2*pi*f2*t); % Sinal 2
sinal3 = sin(2*pi*f3*t); % Sinal 3
```

4.3.3 Geração dos Sinais Individuais

- `f1 = 1;` define a frequência do sinal 1 como 1 Hz.
- `f2 = 3;` define a frequência do sinal 2 como 3 Hz.
- `f3 = 5;` define a frequência do sinal 3 como 5 Hz.
- `sinal1 = sin(2*pi*f1*t);` gera o sinal 1 como uma senoide com frequência de 1 Hz.
- `sinal2 = sin(2*pi*f2*t);` gera o sinal 2 como uma senoide com frequência de 3 Hz.
- `sinal3 = sin(2*pi*f3*t);` gera o sinal 3 como uma senoide com frequência de 5 Hz.

```
% Multiplexa  o TDM
% Definindo o intervalo de tempo para cada sinal
intervalo = 0.1; % 100 ms para cada sinal
amostras_intervalo = round(intervalo * Fs); % N mero de
   amostras por intervalo

% Inicializando o sinal multiplexado
sinal_multiplexado = zeros(size(t));

```

```
% Inserindo cada sinal em sua faixa de tempo
for k = 0:2
    start_idx = k*amostras_intervalo + 1;
    end_idx = min((k+1)*amostras_intervalo, length(t));
    if k == 0
        sinal_multiplexado(start_idx:end_idx) = sinal1(1:(
            end_idx-start_idx+1));
    elseif k == 1
        sinal_multiplexado(start_idx:end_idx) = sinal2(1:(
            end_idx-start_idx+1));
    else
        sinal_multiplexado(start_idx:end_idx) = sinal3(1:(
            end_idx-start_idx+1));
    end
end
```

4.3.4 Multiplexação por Divisão de Tempo (TDM)

- `intervalo = 0.1;` define a duração de 100 ms para cada sinal no processo de multiplexação.
- $amostras_intervalo = round(intervalo * Fs)$; calcula o número de amostras correspondente a 100 ms.
- $sinal_multiplexado = zeros(size(t))$; inicializa o vetor do sinal multiplexado com zeros.
- O loop $fork = 0 : 2$ itera sobre cada sinal para inseri-los no intervalo de tempo apropriado:
  - $start_idx$ e end_idx determinam o intervalo de amostras para cada sinal.
  - Dependendo do valor de `k`, o sinal correspondente (`sinal1`, `sinal2`, ou `sinal3`) é inserido na posição correta do vetor $sinal_multiplexado$.

```
% Demultiplexa  o TDM
% Inicializando os sinais demultiplexados
sinal_demux1 = zeros(size(t));
sinal_demux2 = zeros(size(t));
sinal_demux3 = zeros(size(t));

% Extraindo cada sinal do sinal multiplexado
for k = 0:2
    start_idx = k*amostras_intervalo + 1;
    end_idx = min((k+1)*amostras_intervalo, length(t));
    if k == 0
        sinal_demux1(start_idx:end_idx) = sinal_multiplexado(
           start_idx:end_idx);
    elseif k == 1
        sinal_demux2(start_idx:end_idx) = sinal_multiplexado(
           start_idx:end_idx);
    else
        sinal_demux3(start_idx:end_idx) = sinal_multiplexado(
           start_idx:end_idx);
    end
end
```

4.3.5 Demultiplexação por Divisão de Tempo (TDM)

- $sinal_demux1 = zeros(size(t));$, $sinal_demux2 = zeros(size(t));$, e $sinal_demux3 = zeros(size(t));$ inicializam os vetores para os sinais demultiplexados com zeros.
- O loop $fork = 0 : 2$ itera sobre cada intervalo para extrair os sinais do sinal multiplexado:
  - $start_idx$ e end_idx determinam o intervalo de amostras para cada sinal dentro do vetor $sinal_multiplexado$.
  - Dependendo do valor de `k`, os sinais demultiplexados são preenchidos com os dados extraídos do sinal multiplexado.

```
% Plotagem dos sinais
figure;

subplot(4,1,1);
plot(t, sinal1);
xlabel('Tempo (s)');
ylabel('Amplitude');
title('Sinal 1');

subplot(4,1,2);
plot(t, sinal2);
xlabel('Tempo (s)');
ylabel('Amplitude');
title('Sinal 2');

subplot(4,1,3);
plot(t, sinal3);
xlabel('Tempo (s)');
ylabel('Amplitude');
title('Sinal 3');

subplot(4,1,4);
plot(t, sinal_multiplexado);
xlabel('Tempo (s)');
ylabel('Amplitude');
title('Sinal Multiplexado');

% Plotagem dos sinais demultiplexados
figure;

subplot(3,1,1);
plot(t, sinal_demux1);
xlabel('Tempo (s)');
ylabel('Amplitude');
title('Sinal Demultiplexado 1');

```

```
37 subplot(3,1,2);
38 plot(t, sinal_demux2);
39 xlabel('Tempo (s)');
40 ylabel('Amplitude');
41 title('Sinal Demultiplexado 2');
42
43 subplot(3,1,3);
44 plot(t, sinal_demux3);
45 xlabel('Tempo (s)');
46 ylabel('Amplitude');
47 title('Sinal Demultiplexado 3');
```

4.3.6 Plotagem dos Sinais

- `figure;` cria uma nova figura para a plotagem dos sinais.
- As subplots são usadas para mostrar todos os sinais:
  - `subplot(4,1,1);` plota o `sinal1` no primeiro painel.
  - `subplot(4,1,2);` plota o `sinal2` no segundo painel.
  - `subplot(4,1,3);` plota o `sinal3` no terceiro painel.
  - *subplot*(4,1,4); plota o $sinal_multiplexado$ no quarto painel.
- `figure;` cria uma nova figura para a plotagem dos sinais demultiplexados.
- As subplots são usadas para mostrar os sinais demultiplexados:
  - *subplot*(3,1,1); plota o $sinal_demux1$ no primeiro painel.
  - *subplot*(3,1,2); plota o $sinal_demux2$ no segundo painel.
  - *subplot*(3,1,3); plota o $sinal_demux3$ no terceiro painel.

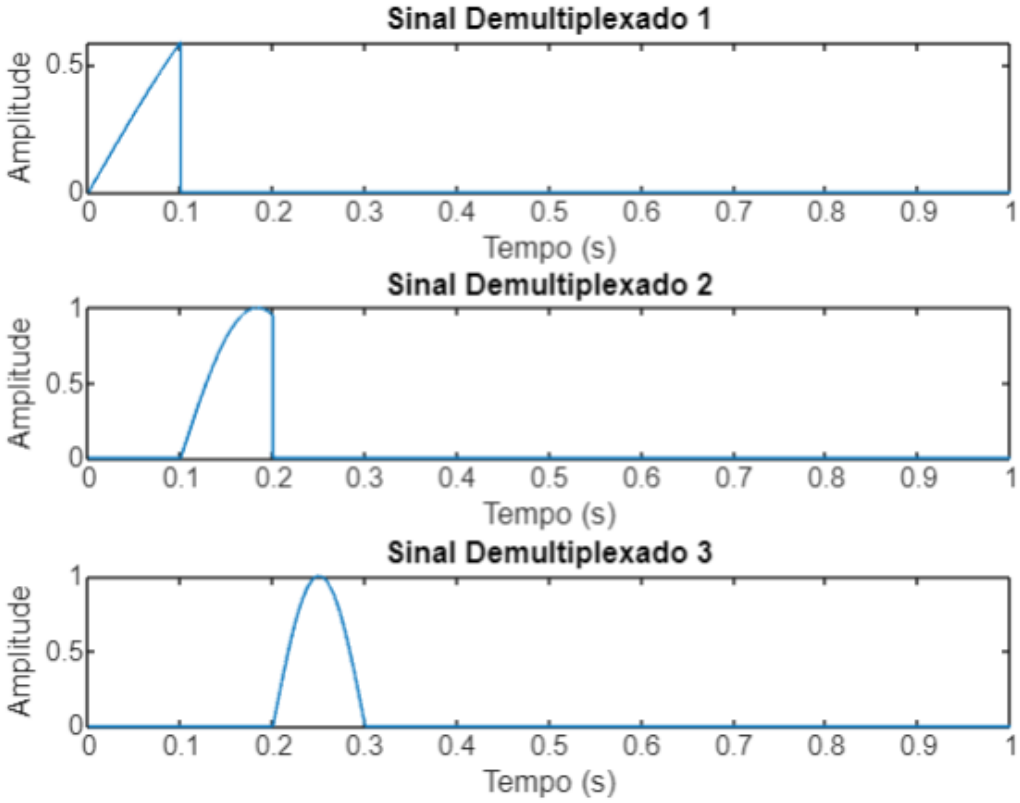


Figura 4 – Fazes da multiplexação. Fonte: Feito pelo próprio autor.

5 Camada de Enlace de Dados

Cuida da transmissão de pacotes de dados entre dois dispositivos conectados diretamente. Ela é responsável pela detecção e correção de erros na camada física, controle de acesso ao meio (MAC) e formação de quadros (frames) (Forouzan e Fegan 2010).

Material: Nesta seção as implementações são feitas em linguagem Python.

5.1 Estudo de técnicas de detecçãao e correção de erros.

A comunicação de dados em redes de computadores está sujeita a diversos tipos de erros, como ruído, interferência, e perda de pacotes, que podem comprometer a integridade dos dados transmitidos. A camada de enlace é responsável por assegurar a transferência confiável de dados entre nós adjacentes na rede, aplicando técnicas de detecção e correção de erros.

5.1.1 Técnicas de Detecção de Erros

A detecção de erros é o processo de identificar se um erro ocorreu durante a transmissão de dados. Algumas das técnicas mais comuns incluem:

5.1.2 Verificação de Redundância Cíclica (CRC)

A Verificação de Redundância Cíclica (CRC) é um método robusto de detecção de erros que usa divisão polinomial. O algoritmo CRC calcula um valor de verificação, que é adicionado ao final dos dados. O receptor realiza a mesma operação para garantir que o valor de verificação coincide com o valor recebido. Se houver um erro na transmissão, o valor de verificação não será correto, indicando a presença de erros.

Listing 5.1 – CRC em Python

```python
def crc_remainder(data, polynomial, initial_remainder=0):
    """Calcula o CRC usando divis o polinomial"""
    data = list(data)  # Convert to list for mutable operations
    poly = list(polynomial)
    remainder = initial_remainder

    for bit in data:
        remainder = (remainder << 1) ^ (int(bit) if bit == '1'
            else 0)
        if remainder & (1 << (len(poly) - 1)):
            remainder ^= int(''.join(poly), 2)

    return bin(remainder)[2:].zfill(len(poly) - 1)

# Teste
data = '1011001'
polynomial = '1101'
remainder = crc_remainder(data, polynomial)
print(f"CRC Remainder: {remainder}")
```

5.1.2.1 Explicação do CRC

- A função $crc_remainder$ calcula o resto da divisão polinomial dos dados pelo polinômio especificado.
- `data` é a string binária dos dados para os quais o CRC será calculado.
- `polynomial` representa o polinômio divisor, também em formato binário.
- `remainder` é o resto inicial, que pode ser ajustado se necessário.
- No loop, o valor do resto é atualizado para cada bit dos dados, e o polinômio é aplicado se o bit mais significativo estiver definido.
- Finalmente, o resto é retornado como o valor CRC, preenchido com zeros à esquerda para garantir o comprimento correto.

5.1.3 Checksum

O Checksum é uma técnica simples de detecção de erros que calcula uma soma dos valores dos dados. O resultado é então usado como uma verificação de integridade. Se o valor calculado no receptor for diferente do valor recebido, pode haver um erro na transmissão.

Listing 5.2 – Checksum em Python

```
def checksum(data):
    """Calcula o checksum simples de uma string bin ria"""
    sum = 0
    for i in range(0, len(data), 8):
        byte = data[i:i+8]
        sum += int(byte, 2)
    return bin(~sum & 0xFF)[2:].zfill(8)

# Teste
data = '10101100'  # Exemplo de string bin ria
checksum_value = checksum(data)
print(f"Checksum: {checksum_value}")
```

5.1.3.1 Explicação do Checksum

- A função `checksum` calcula o valor de checksum para uma string binária.
- O cálculo é feito somando os valores inteiros de cada byte (8 bits) da string binária.
- O resultado é invertido e reduzido a 8 bits usando a operação `sum & 0xFF`.
- O checksum resultante é retornado como uma string binária de 8 bits.

5.1.4 Bits de Paridade

Os Bits de Paridade são usados para detectar erros simples de um único bit. Um bit de paridade é adicionado ao final dos dados para garantir que o número total

de bits '1' seja par (ou ímpar, dependendo do esquema de paridade). O receptor recalcula a paridade e compara com o bit de paridade recebido. Se eles não coincidirem, um erro foi detectado.

Listing 5.3 – Bits de Paridade em Python

```python
def calculate_parity(bits):
    """Calcula o bit de paridade par"""
    count = bits.count('1')
    return '0' if count % 2 == 0 else '1'

def add_parity_bit(data):
    """Adiciona um bit de paridade ao final da string bin ria
       """
    parity_bit = calculate_parity(data)
    return data + parity_bit

def check_parity(data):
    """Verifica o bit de paridade"""
    bits, parity_bit = data[:-1], data[-1]
    return calculate_parity(bits) == parity_bit

# Teste
data = '1010101'
data_with_parity = add_parity_bit(data)
print(f"Data with Parity Bit: {data_with_parity}")
print(f"Parity Check: {check_parity(data_with_parity)}")
```

5.1.4.1 Explicação dos Bits de Paridade

- A função $calculate_parity$ calcula o bit de paridade para uma string binária.
- O bit de paridade é calculado para garantir que o número total de bits '1' seja par.
- A função add_parity_bit adiciona o bit de paridade ao final da string binária.

- A função $check_parity$ verifica se o bit de paridade recebido corresponde ao bit de paridade recalculado.

5.1.5 Código de Hamming

O Código de Hamming é um método de correção de erros que permite detectar e corrigir erros de um único bit. Ele adiciona bits de paridade aos dados para criar um código que pode corrigir erros. O receptor usa esses bits de paridade para identificar e corrigir erros, melhorando a robustez da comunicação.

Listing 5.4 – Código de Hamming em Python

```
1  def hamming_encode(data):
2      """Codifica os dados usando o c digo de Hamming (7,4)"""
3      data_bits = list(data)
4      parity_bits = [0] * 3
5
6      # Calcula os bits de paridade
7      parity_bits[0] = (int(data_bits[0]) + int(data_bits[1]) +
           int(data_bits[3])) % 2
8      parity_bits[1] = (int(data_bits[0]) + int(data_bits[2]) +
           int(data_bits[3])) % 2
9      parity_bits[2] = (int(data_bits[1]) + int(data_bits[2]) +
           int(data_bits[3])) % 2
10
11     return ''.join(map(str, parity_bits + [int(bit) for bit in
           data_bits]))
12
13 def hamming_decode(encoded):
14     """Decodifica os dados usando o c digo de Hamming (7,4)"""
15     parity_bits = [int(encoded[0]), int(encoded[1]), int(
           encoded[2])]
16     data_bits = [int(encoded[3]), int(encoded[4]), int(encoded
           [5]), int(encoded[6])]
17
18     # Calcula os bits de paridade recebidos
19     parity_check = [
```

```
20          (data_bits[0] + data_bits[1] + data_bits[3]) % 2,
21          (data_bits[0] + data_bits[2] + data_bits[3]) % 2,
22          (data_bits[1] + data_bits[2] + data_bits[3]) % 2
23      ]
24
25      # Detecta e corrige erros
26      error_position = parity_check[0] * 1 + parity_check[1] * 2
            + parity_check[2] * 4
27      if error_position:
28          print(f"Erro detectado na posi  o {error_position}")
29          encoded = list(encoded)
30          encoded[error_position - 1] = '1' if encoded[
                error_position - 1] == '0' else '0'
31          print(f"Corrigido: {''.join(encoded)}")
32
33      return ''.join(map(str, encoded[3:]))
34
35  # Teste
36  data = '1011'  # 4 bits de dados
37  encoded = hamming_encode(data)
38  print(f"Encoded with Hamming: {encoded}")
39  decoded = hamming_decode(encoded)
40  print(f"Decoded Data: {decoded}")
```

5.1.5.1 Explicação do Código de Hamming

- A função $hamming_encode$ calcula a codificação Hamming para um conjunto de dados de 4 bits.
- Os bits de paridade são calculados e inseridos nas posições apropriadas para criar um código de 7 bits.
- A função $hamming_decode$ decodifica os dados usando o código de Hamming.
- Verifica se há erros usando os bits de paridade e corrige qualquer erro detectado.
- O erro é localizado pela soma ponderada dos bits de paridade.

5.1.6 Códigos Convolucionais

Diferente dos códigos de bloco, os códigos convolucionais tratam a sequência de bits como um fluxo contínuo. Eles são usados em aplicações onde a transmissão ocorre em tempo real, como em redes sem fio. Neste trabalho não é disponibilizado exemplos de métodos convolucionários.

5.2 Implementação de um protocolo simples de controle de enlace em Python

Um **protocolo de controle de enlace** é um conjunto de regras e procedimentos usados na camada de enlace de dados de uma rede para garantir a transmissão confiável dos dados entre dois dispositivos conectados diretamente, como entre dois computadores em uma rede local ou entre um computador e um roteador. Esses protocolos têm como objetivo lidar com problemas como erros de transmissão, perda de pacotes e a ordenação dos dados, que podem ocorrer durante a comunicação entre os dispositivos (Forouzan e Fegan 2010).

Nesta seção alguns algoritmos são triviais, assim não têm uma explicação ampla, com amostra de variáveis, como em outros capítulos. Apenas um breve texto. Nos casos em que o algoritmo for mais robusto, terá a explicação de forma mais profunda.

5.2.1 Principais Funções dos Protocolos de Controle de Enlace

- **Detecção e Correção de Erros:** Durante a transmissão de dados, é possível que ocorra corrupção devido a interferências ou outros problemas físicos na rede. Protocolos de controle de enlace implementam técnicas como checagem de paridade, CRC (*Cyclic Redundancy Check*), entre outros, para detectar e, em alguns casos, corrigir esses erros.

- **Controle de Fluxo:** O controle de fluxo garante que o transmissor não envie dados mais rápido do que o receptor pode processar. Isso evita o congestionamento do receptor e a perda de pacotes. Um exemplo simples de controle de fluxo é o uso de janelas deslizantes, onde o transmissor envia uma quantidade limitada de pacotes e espera por confirmações (ACKs) antes de enviar mais.

- **Controle de Congestionamento:** Em redes com múltiplos dispositivos, o congestionamento pode ocorrer se muitos pacotes forem enviados ao mesmo tempo.

Protocolos de controle de enlace podem ajustar a taxa de envio para evitar ou aliviar o congestionamento.

- **Retransmissão de Pacotes:** Quando um pacote de dados é perdido ou corrompido, o protocolo de controle de enlace pode solicitar a retransmissão desse pacote. Essa funcionalidade é crucial para garantir que todos os dados cheguem corretamente ao destino.
- **Ordenação de Pacotes:** Em redes onde os pacotes podem chegar fora de ordem (como em redes baseadas em pacotes, como a Internet), o protocolo de controle de enlace garante que os dados sejam reordenados corretamente no destino.

5.2.2 Exemplos de Protocolos de Controle de Enlace

- **Stop-and-Wait:** Um dos protocolos mais simples, onde o transmissor envia um pacote e espera pelo ACK do receptor antes de enviar o próximo pacote. Isso garante que cada pacote é confirmado antes que o próximo seja enviado, mas pode ser ineficiente em redes de alta latência.
- **Sliding Window (Janela Deslizante):** Um protocolo mais eficiente que permite que múltiplos pacotes sejam enviados antes que os ACKs sejam recebidos, utilizando uma janela de pacotes que pode ser ajustada dinamicamente. Isso aumenta a eficiência da transmissão, especialmente em redes de alta latência.
- **Go-Back-N:** Uma variação do Sliding Window, onde, se um pacote é perdido ou corrompido, todos os pacotes subsequentes são retransmitidos, mesmo que alguns tenham sido recebidos corretamente.
- **Selective Repeat:** Outra variação do Sliding Window, onde apenas os pacotes que foram perdidos ou corrompidos são retransmitidos, tornando-o mais eficiente que o Go-Back-N.

Esses protocolos desempenham um papel fundamental na garantia de uma comunicação confiável entre dispositivos em uma rede, minimizando os erros e oti-

mizando o uso dos recursos da rede. Para a prática desta etapa utilizo o conceito de janela deslizante, como pode ser visto na próxima seção.

5.2.3 Explicação do Código

Este código em Python implementa um protocolo simples de controle de enlace com funcionalidades como janela deslizante, retransmissão de pacotes, timeout e controle de congestionamento. Abaixo, explicamos as principais partes do código relacionadas aos conceitos de protocolos de controle de enlace.

5.2.4 Classes e Objetos

- **LinkLayerProtocol:** Esta classe simula o protocolo de controle de enlace, responsável por enviar e receber pacotes, além de lidar com possíveis perdas de pacotes.
- **Sender:** A classe responsável por gerenciar o envio de pacotes, incluindo o controle de fluxo (usando janela deslizante), retransmissão em caso de falha, e ajuste dinâmico do tamanho da janela para simular o controle de congestionamento.
- **Receiver:** A classe responsável por receber pacotes, gerenciar pacotes fora de ordem e enviar confirmações de recebimento (ACKs).

5.2.5 Envio de Pacotes e Simulação de Perda

Na classe `LinkLayerProtocol`, o método `send` simula o envio de um pacote com uma certa probabilidade de perda, definida por `packet_loss_prob`. Se o pacote não for perdido, ele é enviado ao receptor, que chamará o método `receive`:

```
1 def send(self, data, receiver):
2     if random.random() > self.packet_loss_prob:
3         print(f"Enviando pacote: {data}")
4         receiver.receive(data)
```

```
5           return True
6       else:
7           print(f"Pacote perdido: {data}")
8           return False
```

5.2.6 Controle de Janela Deslizante

O controle de janela deslizante é implementado na classe `Sender`. O método `send_data` gerencia a janela de pacotes que está sendo enviada:

```
1  def send_data(self, data):
2      window_start = 0
3      while window_start < len(data):
4          window_end = min(window_start + self.window_size, len(
               data))
5          threads = []
6          for i in range(window_start, window_end):
7              packet = data[i]
8              t = threading.Thread(target=self.
                   send_packet_with_retries, args=(packet, i))
9              threads.append(t)
10             t.start()
11
12         for t in threads:
13             t.join()
14
15         with self.lock:
16             window_start = self.current_ack + 1
17
18         self.adjust_window_size()
```

Aqui, `window_start` e `window_end` definem os limites da janela de pacotes que está sendo enviada. Os pacotes dentro dessa janela são enviados em *threads* separadas, permitindo o envio paralelo.

5.2.7 Retransmissão e Timeout

Se um pacote for perdido ou não for reconhecido dentro de um tempo limite (*timeout*), ele será retransmitido. Isso é gerenciado no método `send_packet_with_retries`:

```
1  def send_packet_with_retries(self, packet, seq_number):
2      attempt = 0
3      while attempt < self.max_retries:
4          success = self.protocol.send((seq_number, packet),
               receiver)
5          if success:
6              if self.wait_for_ack(seq_number):
7                  return
8          attempt += 1
9          print(f"Tentativa {attempt} de {self.max_retries}
               falhou para {packet}. Retentando...")
10     print(f"Falha ao enviar {packet} ap s {self.max_retries}
           tentativas.")
```

Este método tenta enviar o pacote até `max_retries` vezes. Se o pacote for enviado com sucesso, o código espera pelo `ACK` correspondente com `wait_for_ack`. Se o `ACK` não for recebido dentro do tempo limite (*timeout*), o pacote é retransmitido.

5.2.8 Controle de Congestionamento

O controle de congestionamento é simulado no método `adjust_window_size`, que ajusta dinamicamente o tamanho da janela de envio:

```
1  def adjust_window_size(self):
2      with self.lock:
3          if self.window_size < self.congestion_threshold:
4              self.window_size += 1
5          else:
6              self.window_size = max(2, self.window_size // 2)
```

Se o tamanho da janela for menor que o `congestion_threshold`, a janela é aumentada. Caso contrário, a janela é reduzida pela metade, simulando uma resposta

a um possível congestionamento na rede.

5.2.9 Recepção de Pacotes e Gestão de Ordem

O receptor (`Receiver`) gerencia a recepção dos pacotes e lida com pacotes fora de ordem usando um *buffer*:

```
def receive(self, data):
    seq_number, packet = data
    if seq_number == self.expected_seq:
        self.protocol.receive(packet)
        self.expected_seq += 1

        while self.expected_seq in self.buffer:
            buffered_packet = self.buffer.pop(self.expected_seq
                )
            self.protocol.receive(buffered_packet)
            self.expected_seq += 1

    elif seq_number > self.expected_seq:
        print(f"Pacote fora de ordem {packet}, armazenando no
            buffer.")
        self.buffer[seq_number] = packet
```

Se um pacote é recebido fora de ordem, ele é armazenado no *buffer* até que os pacotes anteriores sejam recebidos.

5.2.10 Resumo da implementação

Este código é uma simulação avançada de um protocolo de controle de enlace que incorpora vários conceitos importantes como janela deslizante, retransmissão com *timeout*, controle de congestionamento, e gestão de pacotes fora de ordem. Essas funcionalidades são essenciais para garantir a transmissão confiável de dados em redes onde a perda de pacotes e o congestionamento podem ocorrer.

5.3 Implementação de algoritmos de controle de fluxo

Algoritmos de controle de fluxo são essenciais em redes de comunicação para garantir que os dados sejam transmitidos de forma eficiente e sem sobrecarregar os recursos de rede (Maia 2013).

5.3.1 Stop-and-Wait (Parar e Esperar)

A técnica Stop-and-Wait é um dos métodos mais simples de controle de fluxo. No Stop-and-Wait, o transmissor envia um quadro de dados e aguarda um reconhecimento (ACK) do receptor antes de enviar o próximo quadro. Isso garante que cada quadro seja recebido corretamente antes de prosseguir com o envio do próximo. O processo continua até que todos os quadros sejam enviados e confirmados.

```
import time

def send_frame(frame):
    print(f"Sending frame {frame}")
    time.sleep(1)
    return True

def stop_and_wait(frames):
    for frame in frames:
        while True:
            print(f"Attempting to send frame {frame}")
            if send_frame(frame):
                print(f"ACK received for frame {frame}")
                break
            else:
                print(f"Timeout, resending frame {frame}")
```

5.3.2 Sliding Window (Janela Deslizante)

O algoritmo de janela deslizante permite enviar múltiplos frames antes de precisar de uma confirmação para o primeiro frame na janela. Isso é feito para maximizar a eficiência e utilizar o canal de comunicação de maneira mais eficaz. A janela "desliza"para frente conforme as confirmações (ACKs) são recebidas, permitindo o envio de novos frames enquanto se aguarda a confirmação dos anteriores.

```
import time

def send_frame_sw(frame):
    print(f"Sending frame {frame}")
    time.sleep(1)
    return True

def sliding_window(frames, window_size):
    send_base = 0
    next_frame = 0
    while send_base < len(frames):
        while next_frame < send_base + window_size and next_frame < len(frames):
            send_frame_sw(frames[next_frame])
            next_frame += 1

        ack_received = send_base + 1  # Simulating ACK for the next frame
        print(f"ACK received for frame {ack_received}")
        send_base = ack_received
```

5.3.3 Funções e Variáveis

5.3.4 Função `send_frame_sw(frame)`

- **Propósito:** Simular o envio de um frame e aguardar um segundo antes de retornar um status de sucesso.
- **Parâmetros:** `frame` - o frame a ser enviado.
- **Comportamento:** Imprime uma mensagem indicando que o frame está sendo enviado e aguarda 1 segundo antes de retornar `True`.

5.3.5 Função `sliding_window(frames, window_size)`

- **Propósito:** Implementar o algoritmo de janela deslizante para enviar uma lista de frames com uma janela de tamanho fixo.
- **Parâmetros:**
  - `frames` - uma lista de frames a serem enviados.
  - `window_size` - o tamanho da janela de envio, ou seja, o número máximo de frames que podem ser enviados antes de receber uma confirmação (ACK).
- **Variáveis:**
  - `send_base` - o índice do frame mais baixo que ainda não foi confirmado.
  - `next_frame` - o índice do próximo frame a ser enviado.

5.3.6 Fluxo do Algoritmo

5.3.7 Inicialização

Define `send_base` como 0 e `next_frame` também como 0. Isso significa que o envio começa do primeiro frame.

5.3.8 Loop Principal

- **Objetivo:** Continua enquanto houver frames para enviar.
- **Estrutura:** `while send_base < len(frames)`

5.3.9 Loop Interno

- **Objetivo:** Envia frames até que a janela esteja cheia ou todos os frames tenham sido enviados.
- **Estrutura:** `while next_frame < send_base + window_size and next_frame < len(frames)`
  - Utiliza `send_frame_sw(frames[next_frame])` para enviar o frame atual e incrementa `next_frame`.

5.3.10 Simulação de ACK

- **Objetivo:** Simular a recepção de um ACK (Acknowledgement) para o próximo frame a ser confirmado.
- **Simulação:** Define `ack_received` como `send_base + 1` (o próximo frame após o `send_base`).
- Imprime uma mensagem indicando o recebimento do ACK para o frame `ack_received`.

5.3.11 Atualização do `send_base`

- **Objetivo:** Move a base de envio para o próximo frame confirmado.
- Define `send_base` como o valor de `ack_received`.

5.3.12 Go-Back-N (Voltar-N)

O Go-Back-N é um protocolo de controle de fluxo e correção de erros usado em redes de comunicação. Ele utiliza uma janela deslizante para enviar vários pacotes antes de receber um reconhecimento (ACK) para o primeiro pacote enviado. Caso um erro ocorra ou um ACK se perca, todos os pacotes a partir do ponto do erro devem ser reenviados.

```
import time

def send_frame_gbn(frame):
    print(f"Sending frame {frame}")
    time.sleep(1)
    return True

def go_back_n(frames, window_size):
    send_base = 0
    next_frame = 0
    while send_base < len(frames):
        while next_frame < send_base + window_size and next_frame < len(frames):
            send_frame_gbn(frames[next_frame])
            next_frame += 1

        if next_frame % 3 == 0:
            print(f"ACK lost for frame {send_base + 1}")
        else:
            ack_received = send_base + 1
            print(f"ACK received for frame {ack_received}")
            send_base = ack_received

        if send_base != next_frame:
            print(f"Timeout, resending frames from {send_base}")
```

```
next_frame = send_base
```

5.3.12.0.1 Explicação:

- **Função** `send_frame_gbn(frame)`: Simula o envio de um quadro (frame). Aqui, apenas imprime o número do quadro e faz uma pausa de 1 segundo para simular o tempo de envio.
- **Função** `go_back_n(frames, window_size)`:
  - **Variáveis**:
    - `send_base`: Indica o primeiro quadro da janela de envio que está esperando pelo ACK.
    - `next_frame`: Indica o próximo quadro a ser enviado.
  - **Loop Principal**: Continua enquanto `send_base` for menor que o número total de quadros.
    - **Envio de Quadros**: Envia todos os quadros dentro da janela de tamanho `window_size` e incrementa `next_frame`.
    - **Simulação de Perda de ACK**: A cada 3 quadros enviados, simula a perda do ACK para o quadro atual (`send_base + 1`).
    - **Recebimento de ACK**: Caso o ACK não seja perdido, avança o `send_base` para o próximo quadro confirmado.
    - **Timeout e Reenvio**: Se `send_base` não é igual a `next_frame`, ocorre um timeout e todos os quadros a partir do `send_base` são reenviados.

5.3.13 Selective Repeat (Repetição Seletiva)

O Selective Repeat é um protocolo de controle de fluxo e correção de erros onde apenas os pacotes que falharam são reenviados, ao invés de todos os pacotes após o erro, como no Go-Back-N. Cada pacote é reconhecido individualmente, permitindo uma maior eficiência.

```
import time

def send_frame_sr(frame):
    print(f"Sending frame {frame}")
    time.sleep(1)
    return True

def selective_repeat(frames, window_size):
    send_base = 0
    next_frame = 0
    ack_received = [False] * window_size
    while send_base < len(frames):
        while next_frame < send_base + window_size and next_frame < len(frames):
            send_frame_sr(frames[next_frame])
            next_frame += 1

        ack_number = send_base + (next_frame % window_size)
        ack_received[ack_number % window_size] = True
        print(f"ACK received for frame {ack_number}")

        while send_base < len(frames) and ack_received[send_base % window_size]:
            send_base += 1

        if send_base == next_frame:
            print(f"All frames up to {send_base} acknowledged")
```

5.3.13.0.1 Explicação:

- **Função** `send_frame_sr(frame)`: Simula o envio de um quadro (frame) com uma pausa de 1 segundo para representar o tempo de envio.

- **Função** `selective_repeat(frames, window_size)`:
  - **Variáveis**:
    - `send_base`: Indica o primeiro quadro da janela de envio que está esperando pelo ACK.
    - `next_frame`: Indica o próximo quadro a ser enviado.
    - `ack_received`: Lista que mantém o estado de recebimento dos ACKs para os quadros dentro da janela.
  - **Loop Principal**: Continua enquanto `send_base` for menor que o número total de quadros.
    - **Envio de Quadros**: Envia todos os quadros dentro da janela de tamanho `window_size` e incrementa `next_frame`.
    - **Recebimento de ACK**: Calcula o número do ACK recebido e atualiza a lista `ack_received`.
    - **Avanço de** `send_base`: Avança o `send_base` para os quadros confirmados até o próximo quadro não confirmado.
    - **Confirmação de Todos os Quadros**: Se todos os quadros dentro da janela foram confirmados, imprime a mensagem correspondente.

6 Camada de Rede

Garante que os pacotes de dados sejam entregues ao destinatário correto, mesmo que atravessem várias redes intermediárias. O principal protocolo dessa camada é o IP (Internet Protocol), que define os endereços IP e roteia os pacotes entre diferentes redes (Forouzan e Fegan 2010).

Material: Nesta seção as implementações são feitas em linguagem Python.

6.1 Estudo de protocolos de roteamento como OSPF e BGP

O estudo de protocolos de roteamento como OSPF (Open Shortest Path First) e BGP (Border Gateway Protocol) é fundamental para entender como os dados são direcionados na internet e em redes corporativas. Esses protocolos desempenham papéis distintos, mas complementares, no roteamento de pacotes de dados.

6.1.1 OSPF (Open Shortest Path First)

6.1.2 Tipo de Protocolo

OSPF é um protocolo de roteamento interior (*Interior Gateway Protocol*, IGP) que funciona dentro de uma única rede ou sistema autônomo (AS).

6.1.3 Algoritmo de Roteamento

Utiliza o algoritmo de Dijkstra para calcular o caminho mais curto entre dois pontos. Esse cálculo é baseado no custo das rotas, que normalmente leva em conta a largura de banda, atraso e outros fatores.

6.1.4 Área de Aplicação

Ideal para redes de grande porte, como redes corporativas e redes de provedores de serviço, onde a eficiência no roteamento dentro de um AS é crucial.

6.1.5 Estrutura

OSPF organiza a rede em áreas para melhorar a escalabilidade. A área 0 é a espinha dorsal e conecta todas as outras áreas.

6.1.6 Atualização de Rotas

OSPF atualiza suas tabelas de roteamento rapidamente em resposta a mudanças na topologia da rede, enviando atualizações de estado de link (LSAs) para todos os roteadores na área.

6.1.7 BGP (Border Gateway Protocol)

6.1.8 Tipo de Protocolo

BGP é um protocolo de roteamento exterior (*Exterior Gateway Protocol*, EGP) que funciona entre diferentes sistemas autônomos (AS). É o protocolo que permite a troca de informações de roteamento na internet.

6.1.9 Algoritmo de Roteamento

Utiliza um sistema de roteamento baseado em caminhos e políticas, em vez de calcular o caminho mais curto. As decisões de roteamento em BGP são influenciadas por várias políticas e atributos, como o comprimento do caminho (*AS Path*), a origem da rota, e outras métricas definidas pelos administradores de rede.

6.1.10 Área de Aplicação

Essencial para a conectividade global na internet, interconectando grandes redes, como provedores de serviços de internet (ISPs) e grandes corporações.

6.1.11 Estrutura

BGP não tem uma estrutura de áreas como o OSPF. Em vez disso, funciona em pares de roteadores (*peering*) que trocam informações de roteamento.

6.1.12 Atualização de Rotas

Diferente do OSPF, BGP não atualiza rotas com a mesma frequência. Ele estabelece conexões persistentes para trocar atualizações de roteamento de forma mais esporádica, enviando apenas quando há mudanças significativas na topologia.

6.1.13 Comparação e Uso Conjunto

6.1.14 Escopo

OSPF é usado internamente dentro de uma rede, enquanto o BGP gerencia a comunicação entre redes distintas.

6.1.15 Complexidade

BGP é mais complexo, exigindo uma maior compreensão das políticas de roteamento e das implicações de design da rede.

6.1.16 Flexibilidade

OSPF é mais rápido em se adaptar a mudanças na topologia, enquanto o BGP oferece uma maior flexibilidade em termos de políticas de roteamento e controle.

6.2 Estudo de Técnicas de balanceamento de carga

6.2.1 O que é balanceamento de carga

O balanceamento de carga é o método de distribuir o tráfego de rede igualmente em um grupo de recursos que oferecem suporte a uma aplicação. As aplicações modernas devem processar milhões de usuários simultaneamente e retornar o texto, vídeos, imagens e outros dados corretos para cada usuário de maneira rápida e confiável. Para lidar com volumes tão altos de tráfego, a maioria das aplicações tem muitos servidores de recursos com dados duplicados entre eles. Um balanceador de carga é um dispositivo que fica entre o usuário e o grupo de servidores e atua como um facilitador invisível, garantindo que todos os servidores de recursos sejam usados igualmente (O que é a computação em nuvem? 2024).

6.2.2 Algoritmos de Balanceamento de Carga

Um algoritmo de balanceamento de carga é um conjunto de regras que um balanceador de carga segue para determinar qual servidor deve atender a cada solicitação do cliente. Esses algoritmos podem ser divididos em duas categorias principais:

6.2.3 1. Balanceamento de Carga Estático

Definição: Algoritmos de balanceamento de carga estático seguem regras predefinidas e não dependem do estado atual dos servidores.

Exemplos:

- **Método Round-Robin**: Neste método, os servidores possuem endereços IP que são utilizados para direcionar as solicitações dos clientes. Um sistema de nomes de domínio (DNS) mapeia os nomes dos sites para os endereços IP dos servidores. O balanceamento é realizado pelo servidor de nomes, que retorna os endereços IP dos servidores de forma cíclica (round-robin), sem a necessidade de hardware ou software especializado.

- **Método Round-Robin Ponderado**: Similar ao round-robin, mas com a capacidade de atribuir diferentes pesos aos servidores com base em sua prioridade ou capacidade. Servidores com pesos maiores recebem mais tráfego.

- **Método de Hash IP**: Nesse método, o balanceador de carga executa um cálculo matemático (hash) sobre o endereço IP do cliente. Esse hash é convertido em um número que, por sua vez, é mapeado para um servidor específico.

6.2.4 2. Balanceamento Dinâmico de Carga

Definição: Algoritmos de balanceamento de carga dinâmico analisam o estado atual dos servidores antes de distribuir o tráfego, ajustando-se em tempo real.

Exemplos:

- **Método de Conexão Mínima**: Esse método monitora o número de conexões ativas em cada servidor e direciona o tráfego para o servidor com menos conexões, assumindo que todas as conexões demandam a mesma quantidade de recursos.

- **Método de Conexão Mínima Ponderada**: Similar ao método de conexão mínima, mas considerando que alguns servidores podem suportar mais conexões do que outros. Servidores com maior capacidade ou prioridade recebem mais tráfego, de acordo com o peso atribuído.

- **Método de Menor Tempo de Resposta**: Combina o tempo de resposta do servidor com o número de conexões ativas para determinar qual servidor está mais apto a fornecer um serviço rápido, priorizando o tempo de resposta eficiente.

- **Método Baseado em Recursos**: Aqui, o balanceador de carga distribui o tráfego com base na carga atual de cada servidor. Um software agente em cada servidor monitora o uso de recursos, como CPU e memória, e o balanceador de carga utiliza essas informações para enviar as solicitações para o servidor com recursos livres suficientes.

6.2.5 Algoritmos

6.2.6 Algoritmo Round-Robin

O algoritmo Round-Robin é um método simples e eficiente de balanceamento de carga que distribui as solicitações de forma circular entre um conjunto de servidores. A ideia principal é garantir que todas as instâncias de servidor recebam uma quantidade aproximadamente igual de solicitações, independentemente do estado atual dos servidores.

O algoritmo funciona da seguinte forma:

- Inicialmente, o índice do servidor a ser selecionado é definido como zero.
- Para cada nova solicitação, o servidor correspondente ao índice atual é selecionado.
- Após a seleção, o índice é incrementado para o próximo servidor na lista.
- Quando o índice atinge o final da lista de servidores, ele volta ao início, criando um ciclo contínuo.

Isso garante que cada servidor receba solicitações em uma ordem cíclica, proporcionando uma distribuição equilibrada das solicitações.

O código em Python a seguir implementa o algoritmo Round-Robin:

Listing 6.1 – Algoritmo Round-Robin em Python

```
1 class RoundRobinBalancer:
2     def __init__(self, servers):
3         self.servers = servers
4         self.index = 0
5 
6     def get_next_server(self):
7         server = self.servers[self.index]
8         self.index = (self.index + 1) % len(self.servers)
9         return server
```

```
10
11 # Teste
12 servers = ["Server1", "Server2", "Server3"]
13 rr_balancer = RoundRobinBalancer(servers)
14
15 for _ in range(10):
16     print(f"Server selected: {rr_balancer.get_next_server()}")
```

6.2.6.1 Explicação do Código

- `class RoundRobinBalancer`: Define a classe `RoundRobinBalancer` que gerencia a seleção de servidores utilizando o método Round-Robin.
- $def\_\_init\_\_(self, servers)$: O método $_{init}$ é o construtor da classe, que inicializa a lista de servidores e o índice atual (`self.index`) como zero. A lista de servidores (`self.servers`) é armazenada como um atributo da classe.
- $defget\_next\_server(self)$: O método get_next_server retorna o servidor correspondente ao índice atual (`self.index`). Após retornar o servidor, o índice é incrementado e ajustado para circular de volta ao início da lista se atingir o final. A operação `(self.index + 1) % len(self.servers)` garante que o índice seja sempre um valor válido dentro dos limites da lista de servidores.
- $servers = [\text{"}Server1\text{"}, \text{"}Server2\text{"}, \text{"}Server3\text{"}]$: Define uma lista de servidores para teste.
- $rr\_balancer = RoundRobinBalancer(servers)$: Cria uma instância do balanceador Round-Robin com a lista de servidores definida.
- $for\_inrange(10)$: Um loop que seleciona e imprime o servidor escolhido por 10 iterações para demonstrar o comportamento cíclico do balanceador.

6.2.7 Algoritmo Least Connections

O algoritmo Least Connections é um método de balanceamento de carga que seleciona o servidor com o menor número de conexões ativas. Este método é útil para distribuir a carga de maneira mais equilibrada, considerando o número de conexões atuais em cada servidor. Isso é especialmente importante quando os servidores têm capacidades diferentes e a quantidade de trabalho pode variar.

O funcionamento do algoritmo é o seguinte:

- Inicialmente, todos os servidores são configurados com um contador de conexões ativo igual a zero.
- Quando uma nova solicitação chega, o algoritmo escolhe o servidor com o menor número de conexões ativas.
- Após a escolha, o contador de conexões do servidor selecionado é incrementado.
- Quando uma conexão é finalizada, o contador de conexões do servidor é decrementado.

Esse método garante que o servidor com menor carga atual (em termos de conexões ativas) seja selecionado, promovendo uma distribuição de carga mais equilibrada entre os servidores.

O código em Python a seguir implementa o algoritmo Least Connections:

Listing 6.2 – Algoritmo Least Connections em Python

```
1 class LeastConnectionsBalancer:
2     def __init__(self, servers):
3         self.servers = {server: 0 for server in servers}
4 
5     def get_next_server(self):
6         server = min(self.servers, key=self.servers.get)
7         self.servers[server] += 1
8         return server
```

```
9
10      def release_connection(self, server):
11          if server in self.servers:
12              self.servers[server] -= 1
13
14  # Teste
15  servers = ["Server1", "Server2", "Server3"]
16  lc_balancer = LeastConnectionsBalancer(servers)
17
18  for _ in range(10):
19      server = lc_balancer.get_next_server()
20      print(f"Server selected: {server}")
21      lc_balancer.release_connection(server)
```

6.2.7.1 Explicação do Código

- `class LeastConnectionsBalancer`: Define a classe `LeastConnectionsBalancer` que gerencia a seleção de servidores usando o método Least Connections.
- $def\_\_init\_\_(self, servers)$: O método $_{init}$ é o construtor da classe, que inicializa um dicionário de servidores (`self.servers`) com todos os servidores configurados com um contador de conexões ativo igual a zero.
- $defget\_next\_server(self)$: O método get_next_server seleciona o servidor com o menor número de conexões ativas, utilizando a função `min` para encontrar o servidor com o valor mínimo no dicionário `self.servers`. Após selecionar o servidor, o contador de conexões do servidor é incrementado em um.
- $defrelease\_connection(self, server)$: O método $release\_connection$ é usado para liberar uma conexão em um servidor específico. O contador de conexões do servidor é decrementado em um. Se o servidor não estiver presente no dicionário, o método não realiza nenhuma ação.
- $servers = [\text{"}Server1\text{"}, \text{"}Server2\text{"}, \text{"}Server3\text{"}]$: Define uma lista de servidores para teste.

- $lc\_balancer = LeastConnectionsBalancer(servers)$: Cria uma instância do balanceador Least Connections com a lista de servidores definida.
- $for\_inrange(10)$: Um loop que seleciona e imprime o servidor escolhido por 10 iterações para demonstrar o comportamento do balanceador. Após cada seleção, a conexão é liberada imediatamente para simular um cenário de conexão temporária.

6.2.8 Algoritmo Least Response Time

O algoritmo Least Response Time seleciona o servidor com o menor tempo de resposta registrado. Esse método é útil para garantir que a solicitação seja atendida pelo servidor mais rápido, proporcionando uma melhor experiência ao usuário. O algoritmo considera a eficiência do servidor em responder às solicitações.

O funcionamento do algoritmo é o seguinte:

- Inicialmente, cada servidor é atribuído um tempo de resposta aleatório.
- Quando uma nova solicitação chega, o algoritmo escolhe o servidor com o menor tempo de resposta registrado.
- Após a escolha, o tempo de resposta do servidor selecionado é atualizado com um novo valor simulado, representando o tempo de resposta mais recente.

Esse método ajuda a direcionar as solicitações para o servidor mais eficiente no momento, melhorando a experiência do usuário.

O código em Python a seguir implementa o algoritmo Least Response Time:

Listing 6.3 – Algoritmo Least Response Time em Python

```
1 import random
2
3 class LeastResponseTimeBalancer:
4     def __init__(self, servers):
```

```
5           self.servers = {server: random.uniform(0.5, 1.5) for
                server in servers}
6
7       def get_next_server(self):
8           server = min(self.servers, key=self.servers.get)
9           self.servers[server] = self.get_response_time()
10          return server
11
12      def get_response_time(self):
13          return random.uniform(0.5, 1.5)  # Simula o tempo de
                resposta
14
15  # Teste
16  servers = ["Server1", "Server2", "Server3"]
17  lrt_balancer = LeastResponseTimeBalancer(servers)
18
19  for _ in range(10):
20      print(f"Server selected: {lrt_balancer.get_next_server()}")
```

6.2.8.1 Explicação do Código

- `import random`: Importa o módulo `random`, que é usado para gerar tempos de resposta aleatórios.
- `class LeastResponseTimeBalancer`: Define a classe `LeastResponseTimeBalancer` que gerencia a seleção de servidores usando o método Least Response Time.
- $def\_\_init\_\_(self, servers)$: O método $_{init}$ é o construtor da classe, que inicializa um dicionário de servidores (`self.servers`) com tempos de resposta aleatórios, variando de 0.5 a 1.5 segundos.
- $defget\_next\_server(self)$: O método get_next_server seleciona o servidor com o menor tempo de resposta registrado, utilizando a função `min` para encontrar o servidor com o menor valor no dicionário `self.servers`. Após selecionar

o servidor, o tempo de resposta do servidor é atualizado com um novo valor simulado retornado pelo método `get_response_time`.

- $def get\_response\_time(self)$: O método $get\_response\_time$ simula o tempo de resposta do servidor retornando um valor aleatório entre 0.5 e 1.5 segundos.
- $servers = [\text{"}Server1\text{"}, \text{"}Server2\text{"}, \text{"}Server3\text{"}]$: Define uma lista de servidores para teste.
- $lrt\_balancer = LeastResponseTimeBalancer(servers)$: Cria uma instância do balanceador Least Response Time com a lista de servidores definida.
- $for\_inrange(10)$: Um loop que seleciona e imprime o servidor escolhido por 10 iterações para demonstrar o comportamento do balanceador.

6.3 Implementação de algoritmos de roteamento em Python

Para implementar um algoritmo de roteamento em Python, foi feito uma versão simplificada do algoritmo de roteamento de Dijkstra, que encontra o caminho mais curto entre dois nós em um grafo. Esse algoritmo é amplamente utilizado em redes para determinar a rota mais eficiente entre pontos (Forouzan e Fegan 2010).

6.3.1 Implementação do Algoritmo de Dijkstra

```
import heapq

def dijkstra(grafo, origem):
    distancias = {no: float('infinity') for no in grafo}
    distancias[origem] = 0
    fila_prioridade = [(0, origem)]

    while fila_prioridade:
        distancia_atual, no_atual = heapq.heappop(fila_prioridade)
        if distancia_atual > distancias[no_atual]:
            continue
        for vizinho, peso in grafo[no_atual].items():
            distancia = distancia_atual + peso
            if distancia < distancias[vizinho]:
                distancias[vizinho] = distancia
                heapq.heappush(fila_prioridade, (distancia, vizinho))

    return distancias

grafo = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
```

```
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

distancias = dijkstra(grafo, 'A')
print(distancias)
```

6.3.2 Como o Algoritmo Funciona

6.3.3 Inicialização

O algoritmo começa inicializando as distâncias de todos os nós como infinito, exceto para o nó de origem, cuja distância é zero.

6.3.4 Fila de Prioridade

Utilizamos uma fila de prioridade (implementada com `heapq`) para sempre selecionar o nó com a menor distância conhecida.

6.3.5 Atualização das Distâncias

Para cada nó, o algoritmo verifica seus vizinhos. Se o caminho através do nó atual for mais curto do que a distância conhecida anteriormente, a distância é atualizada e o vizinho é adicionado à fila de prioridade.

6.3.6 Resultado

No final, o algoritmo retorna um dicionário com as menores distâncias da origem para todos os outros nós no grafo.

6.3.7 Exemplo de Saída

Com o grafo fornecido, a saída será:

```
{'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

Isso significa que a distância mais curta do nó `'A'` para `'B'` é 1, para `'C'` é 3, e para `'D'` é 4.

7 Camada de Transporte

Fornece a transferência confiável de dados de ponta a ponta entre sistemas. Ela é responsável pelo controle de fluxo, correção de erros e multiplexação dos dados para diferentes aplicações. Os principais protocolos dessa camada são o TCP (Transmission Control Protocol) e o UDP (User Datagram Protocol) (Forouzan e Fegan 2010).

Material: Wireshark e implementações em Python.

7.1 Captura e análise de pacotes TCP/UDP com Wireshark

A Internet funciona, em grande parte, através de um modelo de requisições e respostas. Um exemplo claro disso é a navegação na Web: ao usar um navegador, como Google Chrome ou Mozilla Firefox, fazemos uma requisição a um servidor de um determinado site. Se tudo correr bem, o servidor responde com a página web que estávamos procurando.

Para que essa comunicação ocorra de forma eficiente, a Internet utiliza protocolos de transmissão de dados. Dois dos principais protocolos utilizados são o TCP (Transmission Control Protocol) e o UDP (User Datagram Protocol). Ambos fazem parte do conjunto de protocolos da Internet (TCP/IP), mas operam de maneiras distintas e são escolhidos de acordo com as necessidades específicas de diferentes tipos de aplicações, a 10 ilustra bem a comunicação.

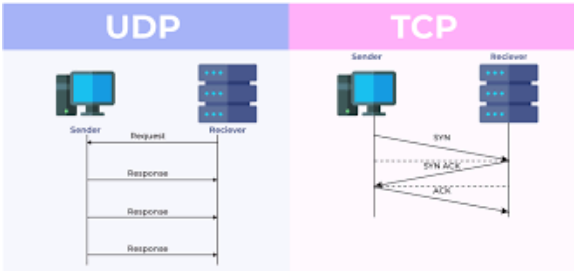


Figura 5 – UDPxTCP. Fonte: (Neves 2024).

7.1.1 TCP (Transmission Control Protocol)

7.1.2 Confiabilidade

O TCP é um protocolo orientado à conexão. Isso significa que, antes de qualquer dado ser transmitido, uma conexão confiável é estabelecida entre o cliente e o servidor, utilizando um processo chamado *three-way handshake*. O TCP garante que todos os pacotes enviados cheguem ao destino e na ordem correta. Se algum pacote se perder ou chegar corrompido, ele será retransmitido.

7.1.3 Controle de Fluxo e Congestionamento

O TCP ajusta dinamicamente a taxa de envio de dados para evitar congestionamento na rede. Ele usa mecanismos de controle de fluxo para garantir que o remetente não sobrecarregue o receptor com mais dados do que ele pode processar.

7.1.4 Sequenciamento e Verificação de Erros

Cada pacote enviado via TCP é numerado, e o receptor os reordena se chegarem fora de ordem. O TCP também verifica a integridade dos dados usando somas de verificação (*checksums*), retransmitindo pacotes se forem detectados erros.

7.1.5 Uso Comum

O TCP é utilizado em aplicações onde a confiabilidade é crucial, como navegação na web (HTTP/HTTPS), transferência de arquivos (FTP), e-mail (SMTP), entre outros.

7.1.6 UDP (User Datagram Protocol)

7.1.7 Não Orientado à Conexão

O UDP é um protocolo sem conexão. Ele envia pacotes chamados "datagramas"sem primeiro estabelecer uma conexão com o destinatário. Não há garantia de

que os pacotes cheguem ao destino, nem que cheguem na ordem correta.

7.1.8 Menor Sobrecarga

O UDP tem menos sobrecarga que o TCP, porque não realiza controle de fluxo, retransmissão automática de pacotes, nem garante a ordem dos pacotes. Isso resulta em uma transmissão de dados mais rápida, porém menos confiável.

7.1.9 Sem Controle de Fluxo e Congestionamento

Como não há controle de fluxo ou congestionamento, o UDP simplesmente envia pacotes o mais rápido que a rede permitir, o que pode levar à perda de pacotes se a rede estiver congestionada.

7.1.10 Uso Comum

O UDP é ideal para aplicações onde a velocidade é mais importante que a confiabilidade, como streaming de vídeo ou áudio, jogos online, DNS (Domain Name System), VoIP (Voice over IP), e outras aplicações em tempo real onde perder alguns pacotes não é crítico.

7.1.11 As principais diferenças

- **Confiabilidade**: O TCP garante a entrega e a ordem dos pacotes; o UDP não.
- **Conexão**: O TCP é orientado à conexão; o UDP é sem conexão.
- **Velocidade**: O UDP é geralmente mais rápido devido à menor sobrecarga; o TCP é mais lento, mas garante a integridade dos dados.
- **Uso**: O TCP é usado para aplicações que requerem alta confiabilidade; o UDP é usado para aplicações que exigem baixa latência e podem tolerar alguma perda de dados.

7.1.12 Análise com Wireshark

Adicionado o filtro de UDP no Wireshark, podemos ver as seguintes requisições:

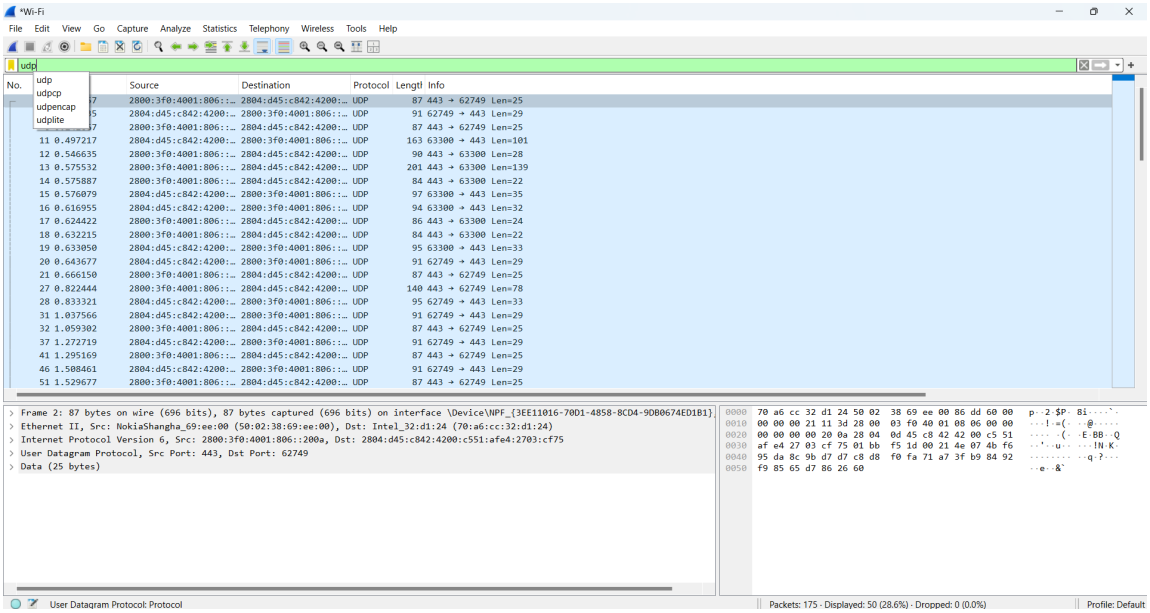


Figura 6 – aaa. Fonte: Próprio autor.

Podemos ver que houveram várias mensagens trocadas por UDP. Temos a aba 'sourse' e a aba 'Destination', nos quais a primeira representa a fonte e a segunda o endereço de envio. Depois vem a aba de 'prococol', que está com o filtro de UDP, logo todas as respostas esão em UDP. Na sequência temos o 'Length', que representa o tamanho da mensagem enviada e por fim, tem a 'info' representando a porta de saída e a de chegada. Em seguida foi aberto a aba de mais informações, as quais indica que há uma porta mandando informação e uma recebendo a informação. Não tem mais nenhuma informação, como esperado. Visto que as mensagens UDP não têm passos de segurança e criptografia.

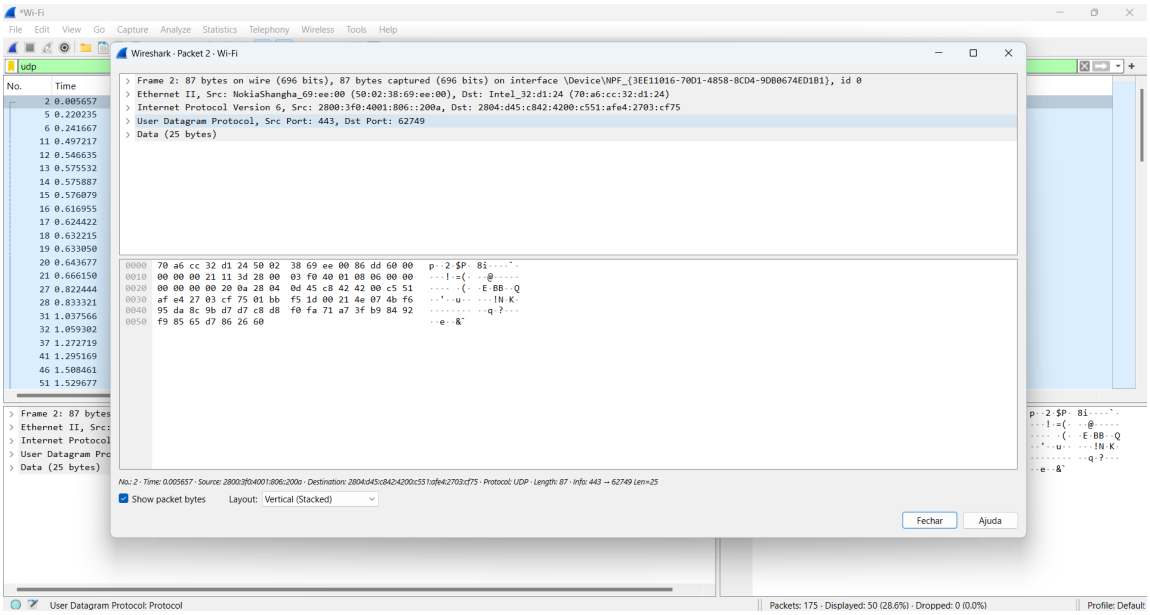


Figura 7 – aaa. Fonte: Próprio autor.

Aqui podemos ver mais detalhes sobre a troca de mensagens UDP. Há 5 campos, os quais representam:

7.1.13 Frame

A seção **Frame** contém informações sobre o próprio pacote de dados na camada de enlace. As informações incluídas são:

- **Número do Pacote**: Identifica o pacote na sequência de captura.
- **Timestamp**: Marca o momento exato em que o pacote foi capturado.
- **Comprimento do Pacote**: O tamanho total do pacote, em bytes.
- **Protocolo de Enlace**: O protocolo utilizado na camada de enlace, por exemplo, Ethernet.

7.1.14 Ethernet II

Se a captura é de uma rede Ethernet, a seção **Ethernet II** fornece informações sobre o cabeçalho Ethernet, incluindo:

- **Endereço MAC de Origem**: O endereço físico do dispositivo que enviou o pacote.
- **Endereço MAC de Destino**: O endereço físico do dispositivo que deve receber o pacote.
- **Tipo de Protocolo de Camada Superior**: Indica o protocolo da camada superior, como IPv4 ou IPv6.

7.1.15 Internet Protocol Version 4 (IPv4) ou Internet Protocol Version 6 (IPv6)

Esta seção fornece informações sobre o cabeçalho IP do pacote:

- **IPv4**:
  - **Endereço IP de Origem**: O endereço IP do remetente.
  - **Endereço IP de Destino**: O endereço IP do destinatário.
  - **Tempo de Vida (TTL)**: O tempo que o pacote pode permanecer na rede antes de ser descartado.
  - **Protocolo de Camada Superior**: O protocolo da camada superior, como TCP ou UDP.
  - **Informações sobre Fragmentação**: Detalhes sobre a fragmentação do pacote, se aplicável.
- **IPv6**:
  - **Endereço IP de Origem**: O endereço IP do remetente.

  - **Endereço IP de Destino**: O endereço IP do destinatário.
  - **Classe de Tráfego**: Informações sobre a prioridade do tráfego.
  - **Fluxo**: Identificador de fluxo para gerenciar pacotes em um fluxo de dados.

7.1.16 Transmission Control Protocol (TCP) ou User Datagram Protocol (UDP)

Dependendo do protocolo de transporte utilizado, esta seção mostra o cabeçalho correspondente:

- **TCP**:
  - **Portas de Origem e Destino**: As portas de comunicação no remetente e no destinatário.
  - **Número de Sequência e Confirmação**: Usado para rastrear a ordem dos pacotes e confirmar a recepção.
  - **Comprimento da Janela**: O tamanho da janela de recepção, que controla o fluxo de dados.
  - **Flags TCP**: Indicadores de controle como SYN e ACK.
- **UDP**:
  - **Portas de Origem e Destino**: As portas de comunicação no remetente e no destinatário.
  - **Comprimento do Pacote UDP**: O tamanho do pacote UDP, em bytes.

7.1.17 Data

A seção **Data** exibe os dados do payload do pacote. Pode incluir:

- **Conteúdo da Mensagem**: O corpo de uma mensagem, como o conteúdo de um e-mail.

- **Conteúdo de uma Solicitação HTTP**: Detalhes sobre uma solicitação ou resposta HTTP.
- **Outros Dados Úteis**: Qualquer outro dado transportado pelo pacote.

Esses campos aparecem tanto para inspeções em UDP quanto para TCP.

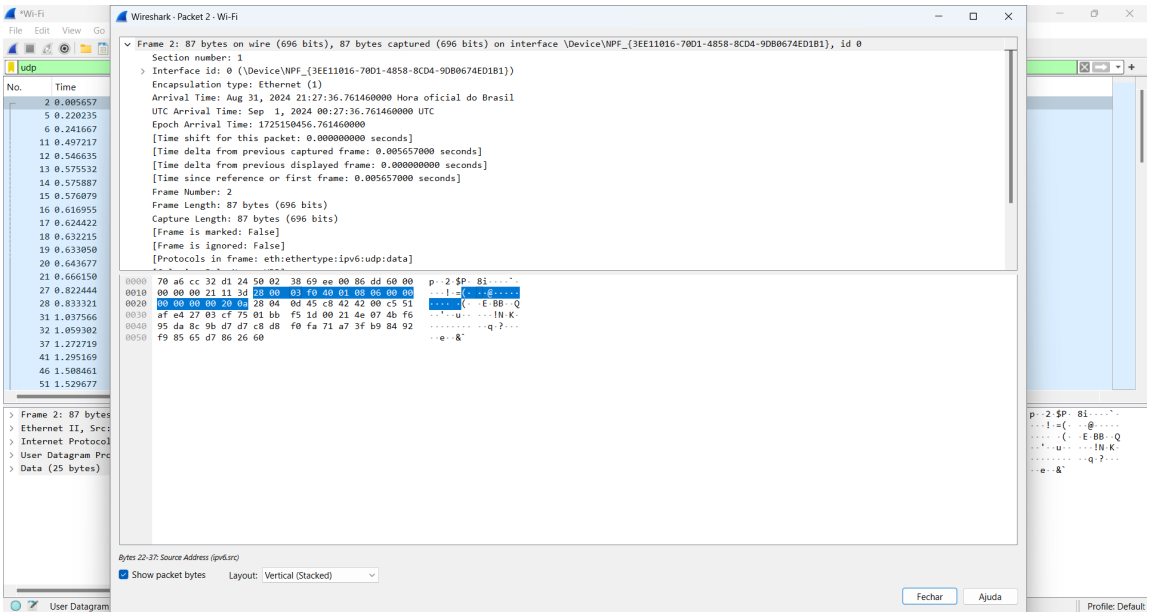


Figura 8 – aaa. Fonte: Próprio autor.

Ao selecionar a aba **Frame**, você pode visualizar as seguintes informações sobre o pacote de dados:

- **Número do Pacote**: Identifica o pacote dentro da captura, permitindo referência e organização.
- **Timestamp**: Mostra a data e hora exatas em que o pacote foi capturado, útil para análise temporal.
- **Comprimento do Pacote**: Indica o tamanho total do pacote em bytes, abrangendo tanto o cabeçalho quanto os dados.

- **Protocolo de Enlace**: Especifica o tipo de protocolo da camada de enlace utilizado para o pacote, como Ethernet.
- **Número do Pacote**: Detalha informações adicionais sobre o pacote, incluindo bytes e parâmetros relevantes.

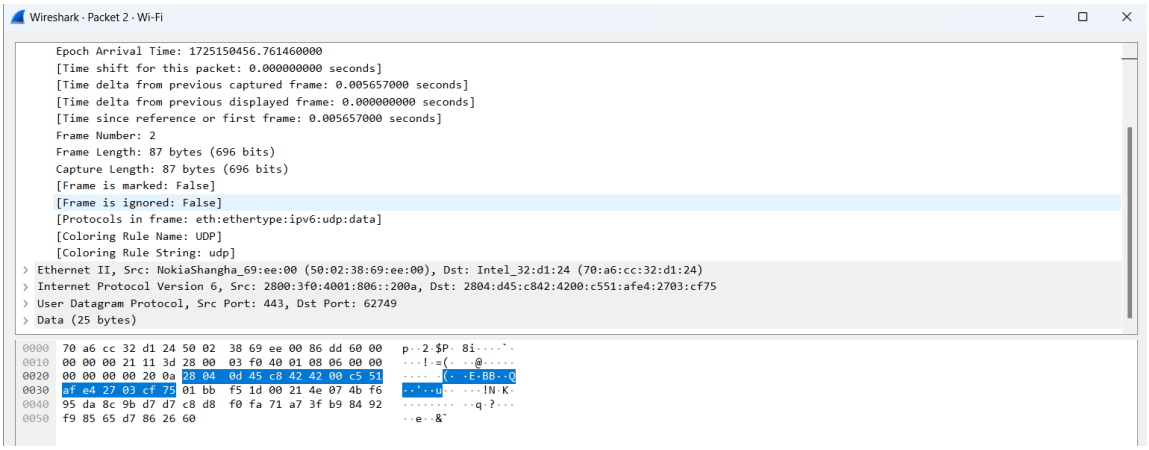


Figura 9 – aaa. Fonte: Próprio autor.

Além da aba **Frame**, ao selecionar as outras opções, você pode visualizar as seguintes informações:

7.1.18 Ethernet II

- **Endereço MAC de Origem**: O endereço MAC do dispositivo que enviou o pacote.
- **Endereço MAC de Destino**: O endereço MAC do dispositivo que deve receber o pacote.
- **Tipo de Protocolo**: Indica o tipo de protocolo da camada superior, como IPv4 ou IPv6.

7.1.19 Internet Protocol Version 4 (IPv4) ou Internet Protocol Version 6 (IPv6)

- **Endereço IP de Origem**: O endereço IP do dispositivo que enviou o pacote.
- **Endereço IP de Destino**: O endereço IP do dispositivo que deve receber o pacote.
- **Tempo de Vida (TTL)**: Em IPv4, o TTL indica a quantidade de saltos que o pacote pode fazer antes de ser descartado.
- **Protocolo de Camada Superior**: Indica o protocolo de transporte usado, como TCP ou UDP.
- **Classe de Tráfego e Fluxo**: Em IPv6, esses campos são usados para gerenciar a qualidade do serviço e o fluxo de pacotes.

7.1.20 Transmission Control Protocol (TCP) ou User Datagram Protocol (UDP)

- **Portas de Origem e Destino**: Identificam os endpoints da comunicação de rede.
- **Número de Sequência e Confirmação**: Em TCP, esses números são usados para rastrear a ordem e a entrega dos pacotes.
- **Comprimento da Janela**: Em TCP, indica a quantidade de dados que o receptor está disposto a aceitar.
- **Flags TCP**: Mostra os flags como SYN, ACK, usados para controle da conexão.
- **Comprimento do Pacote UDP**: Indica o tamanho total do pacote UDP.

7.1.21 Data

- **Conteúdo do Payload**: Exibe os dados transportados pelo pacote, como o corpo de uma mensagem ou o conteúdo de uma solicitação HTTP.

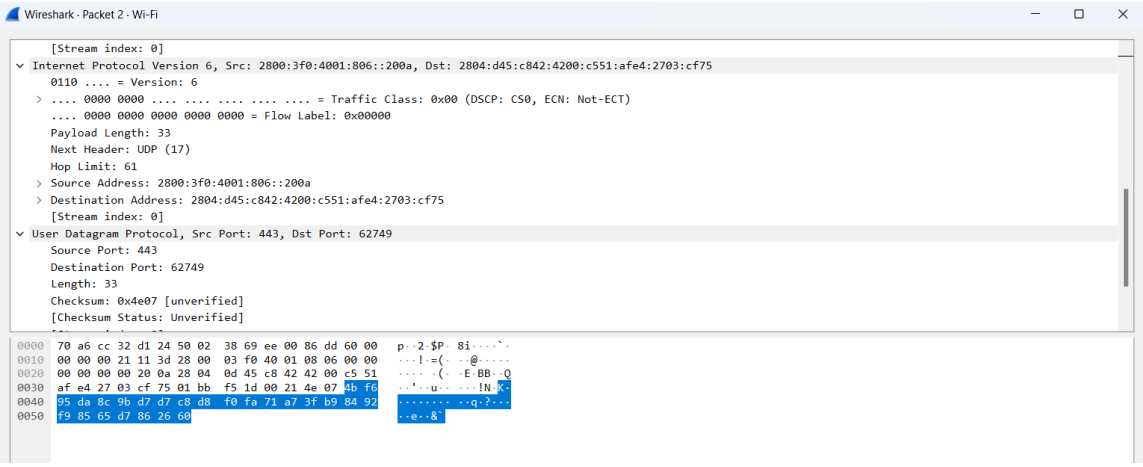


Figura 10 – aaa. Fonte: Próprio autor.

Aqui termina a inspeção do UDP, ressaltando que as informações aparecem tanto em UDP quanto em TCP, sendo assim, as definições da primeira parte servem para a segunda.

Aqui começa-se a leitura das informações TCP:

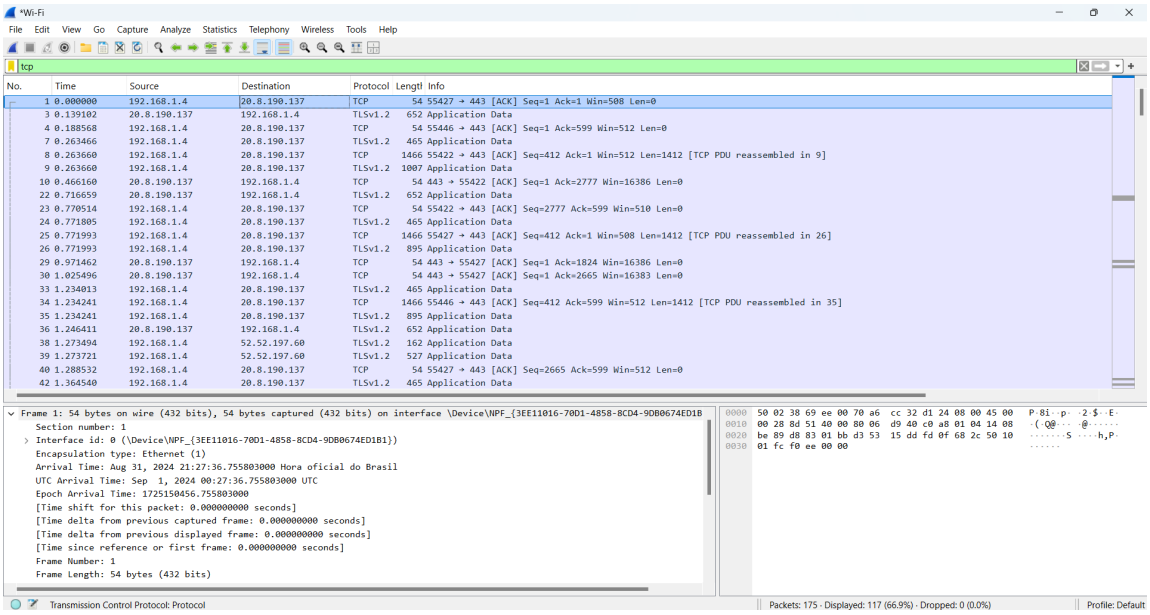


Figura 11 – Fonte: Próprio autor.

Por fim, podemos ver, comparando as imagens que nas conexões UDP o servidor mandam para o Cliente e o cliente manda para o servidor, e não há nenhum outro passo no meio, como podemos ver na imagem 6. Já nas conexões em TCP há alguns passos no meio, como podemos ver na imagem 11, nela no campo 'INFO' é possível ver que a algumas descrições, como: [ACK], onde mostra algumas informações, basicamente essas informações são para controle de tráfego.

7.2 Implementação de algoritmos de controle de congestionamento

A principal causa de descarte de pacotes na rede é o congestionamento. O protocolo TCP, parte do conjunto TCP/IP, garante a transmissão confiável de dados através de um mecanismo de reconhecimento e retransmissão 2. Cada pacote enviado é confirmado por um ACK (Acknowledgement) que informa o número sequencial do próximo pacote esperado. Se o ACK não for recebido dentro de um temporizador, o TCP retransmite o pacote. A fórmula para o temporizador é dada por:

$$\text{temporizador} = b \times R$$

onde:

- R é a estimativa do RTT (Round Trip Time), que representa o tempo decorrido entre a transmissão de um pacote e o recebimento do ACK.
- b é um fator de multiplicação para o temporizador, normalmente definido como 2.

Três ACKs duplicados também indicam que o pacote foi perdido e devem ser retransmitidos.

Para mitigar o impacto do alto atraso de transmissão, o TCP usa uma ”janela deslizante”, que permite enviar múltiplos pacotes antes de receber ACKs. O tamanho da janela ajusta-se conforme os ACKs são recebidos, e controla o fluxo para não sobrecarregar o destinatário.

O congestionamento ocorre quando há mais pacotes na rede do que a capacidade de processamento dos roteadores. Isso pode ser causado por vários fatores, como:

- Filas de roteadores cheias, que levam ao descarte de pacotes.

- Memória insuficiente nos roteadores, que resulta no descarte de pacotes.
- Processadores lentos que causam atraso no processamento dos pacotes.

Se os roteadores têm memória limitada, pacotes serão descartados, e pacotes duplicados podem aumentar a carga na rede. Mesmo com memória suficiente, processadores lentos podem causar congestionamento devido à lentidão no processamento dos pacotes (Andrade *et al.* 2003).

O termo controle de congestionamento, introduzido na Internet no final da década de 80 por 3, é usado para descrever os esforços realizados pelos nós da rede para impedir ou responder a condições de sobrecarga. O controle de congestionamento do TCP é realizado por quatro algoritmos: Slow Start, Congestion Avoidance, Fast Retransmit e Fast Recovery. Apesar de serem independentes, esses algoritmos são geralmente implementados de forma conjunta.

7.2.1 Teoria do Algoritmo

O *Algoritmo de Controle de Congestionamento de Janela Deslizante* é usado para regular a quantidade de dados que um emissor pode enviar antes de receber uma confirmação do receptor. O objetivo é evitar que a rede fique sobrecarregada e garantir que a transmissão de dados ocorra de forma eficiente. A teoria por trás desse algoritmo se baseia em dois estados principais:

- **Slow Start**: O emissor começa com uma janela de congestionamento pequena e dobra o tamanho da janela a cada confirmação recebida, até atingir um limiar chamado *ssthresh*.
- **Congestion Avoidance**: Após atingir o *ssthresh*, a janela de congestionamento aumenta de forma mais lenta e linear, para evitar sobrecarga da rede.

7.2.2 Implementação do Algoritmo em Python

A seguir está a implementação do algoritmo em Python, seguido de uma explicação detalhada de cada parte do código.

7.2.3 Código Python

```
import random
import time

class CongestionControl:
    def __init__(self, max_window_size):
        self.window_size = 1  # Tamanho inicial da janela
        self.max_window_size = max_window_size  # Tamanho máximo da janela
        self.ssthresh = max_window_size / 2  # Limiar de transição para a fase de con
        self.state = 'slow_start'  # Estado inicial
        self.data_sent = 0  # Dados enviados
        self.data_acked = 0  # Dados confirmados

    def send_data(self):
        print(f"Enviando {self.window_size} pacotes")
        self.data_sent += self.window_size

    def receive_ack(self):
        acks = random.randint(1, self.window_size)
        self.data_acked += acks
        print(f"Recebido {acks} confirmações")

    def adjust_window(self):
        if self.state == 'slow_start':
            if self.data_acked >= self.window_size:
```

```
                self.window_size = min(self.window_size * 2, self.max_window_size)
                self.data_acked = 0
                if self.window_size >= self.ssthresh:
                    self.state = 'congestion_avoidance'
                print(f"Janela aumentada para {self.window_size} (slow start)")
            else:
                print("Aguardando confirmações suficientes")
        elif self.state == 'congestion_avoidance':
            if self.data_acked >= self.window_size:
                self.window_size += 1
                self.data_acked = 0
                print(f"Janela aumentada para {self.window_size} (congestion avoidanc

    def simulate(self, rounds):
        for _ in range(rounds):
            self.send_data()
            self.receive_ack()
            self.adjust_window()
            time.sleep(1)  # Simula o tempo de rede

max_window_size = 16
congestion_control = CongestionControl(max_window_size)
congestion_control.simulate(rounds=10)
```

7.2.4 Explicação do Código

- **Classe** `CongestionControl`: Define a classe que simula o controle de congestionamento.
- **Método** $_{init}$: Inicializa os parâmetros do controle de congestionamento, como o tamanho da janela inicial e o tamanho máximo da janela. Define o estado

inicial como *slow start*.

- **Método** `send_data`: Simula o envio de pacotes com base no tamanho da janela de congestionamento.
- **Método** `receive_ack`: Simula o recebimento de confirmações de pacotes, atualizando o número de pacotes confirmados.
- **Método** `adjust_window`: Ajusta o tamanho da janela de congestionamento com base no estado atual e nas confirmações recebidas. Em *slow start*, a janela dobra de tamanho até atingir o limiar *ssthresh*. Em *congestion avoidance*, a janela aumenta linearmente.
- **Método** `simulate`: Executa a simulação do controle de congestionamento por um número definido de rodadas.

7.2.5 Aplicações do Algoritmo

O algoritmo de controle de congestionamento é amplamente utilizado em protocolos de rede, como o TCP, para gerenciar a quantidade de dados transmitidos e garantir uma comunicação eficiente. Suas principais aplicações incluem:

- **Redes de Computadores**: Regula o tráfego de dados para evitar congestionamento e garantir a eficiência da rede.
- **Protocolos de Comunicação**: Implementado em protocolos como o TCP para ajustar dinamicamente a taxa de envio de pacotes.
- **Sistemas de Transferência de Dados**: Utilizado em sistemas que precisam otimizar a transferência de dados entre diferentes partes da rede.

7.3 Desenvolvimento de aplicações cliente-servidor usando sockets em Python

A arquitetura cliente-servidor é um modelo fundamental na infraestrutura de redes, amplamente adotado em ambientes de TI. Nesta abordagem, os computadores são categorizados em duas funções principais: servidores e clientes. Servidores são responsáveis por fornecer serviços e recursos, enquanto clientes solicitam esses serviços e recursos, recebendo as respostas correspondentes, como pode ser visto na imagem 12.

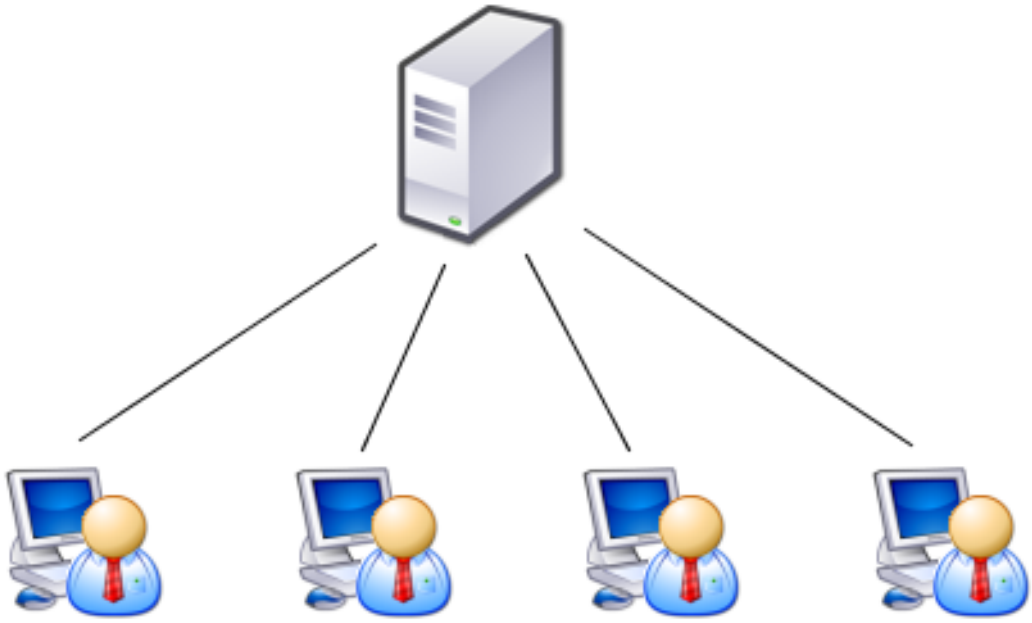


Figura 12 – Cliente-Servidor. Fonte: próprio autor.

A comunicação entre clientes e servidores é realizada através de protocolos de rede, como TCP/IP, utilizando a internet ou redes privadas. Esse modelo é altamente escalável, flexível e seguro, permitindo que as empresas distribuam seus serviços e recursos em diversos dispositivos e plataformas.

Entre as principais vantagens da arquitetura cliente-servidor estão a escalabilidade, segurança e confiabilidade. Esses atributos fazem com que seja uma escolha preferencial para a implementação de redes em larga escala. A estrutura é composta por uma variedade de componentes, como redes, protocolos de comunicação, servidores, clientes e bancos de dados, oferecendo um ambiente personalizável que pode ser ajustado para atender às necessidades específicas de diferentes empresas.

7.3.1 Servidor

O servidor escuta por conexões em uma porta específica e, uma vez conectado, pode receber e enviar dados.

```
1  import socket
2
3  HOST = '127.0.0.1'
4  PORT = 65432
5
6  with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
7      s.bind((HOST, PORT))
8      s.listen()
9      print(f'Servidor escutando em {HOST}:{PORT}')
10
11     conn, addr = s.accept()
12     with conn:
13         print(f'Conectado por {addr}')
14         while True:
15             data = conn.recv(1024)
16             if not data:
17                 break
18             print(f'Recebido: {data.decode()}')
19             conn.sendall(data)
```

1. **Importação do Módulo Socket**

- `import socket` Importa o módulo `socket`, que fornece acesso às funcionalidades de rede do sistema operacional.

2. **Configuração de Endereço e Porta**

   - `HOST = '127.0.0.1'` Define o endereço IP no qual o servidor irá escutar. Neste caso, é o endereço local (localhost).
   - `PORT = 65432` Define a porta na qual o servidor irá escutar por conexões. A porta deve ser um número disponível e não utilizado por outros serviços.

3. **Criação e Configuração do Socket**

- `with socket.socket(socket.AF`$_I NET, socket.SOCK_S TREAM) ass : Criaumobjeto$`socketuti`

4. **Aceitação e Manipulação de Conexões**

   - `conn, addr = s.accept()`
   - Aceita uma conexão de um cliente. O método `accept()` retorna um novo socket para a comunicação com o cliente e o endereço do cliente.
   - `with conn:` Garante que o socket de conexão seja fechado automaticamente após o término da comunicação.
   - `print(f'Conectado por addr')` Imprime uma mensagem indicando que um cliente se conectou, mostrando o endereço do cliente.
   - `while True:` Inicia um loop contínuo para receber e processar dados do cliente.
   - `data = conn.recv(1024)` Recebe até 1024 bytes de dados enviados pelo cliente.
   - `if not data: break` Verifica se não há mais dados (conexão fechada pelo cliente). Se não houver dados, o loop é encerrado.
   - `print(f'Recebido: data.decode()')` Decodifica e imprime os dados recebidos do cliente.
   - `conn.sendall(data)` Envia os dados recebidos de volta para o cliente, realizando um eco.

7.3.2 Cliente

O cliente se conecta ao servidor e pode enviar e receber dados.

```
import socket

HOST = '127.0.0.1'
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Ol , servidor')
    data = s.recv(1024)

print(f'Recebido do servidor: {data.decode()}')
```

1. **Importação do Módulo Socket**

   - `import socket` Importa o módulo `socket`, que fornece acesso às funcionalidades de rede do sistema operacional.

2. **Definição do Endereço e Porta**

   - `HOST = '127.0.0.1'` Define o endereço IP do servidor ao qual o cliente se conectará. Neste caso, `'127.0.0.1'` refere-se ao endereço local (localhost).
   - `PORT = 65432` Define a porta na qual o servidor está escutando. O cliente usará essa porta para se conectar ao servidor.

3. **Criação e Configuração do Socket**

- `with socket.socket(socket.AF`$_I NET, socket.SOCK_S TREAM) as : Criaumobjetosocketus$

4. **Conexão ao Servidor**

   - `s.connect((HOST, PORT))` Estabelece uma conexão com o servidor no endereço IP e na porta especificados.

5. **Envio de Dados para o Servidor**

   - `s.sendall(b'Olá, servidor')` Envia a mensagem `b'Olá, servidor'` para o servidor. O prefixo `b` indica que a string está no formato de bytes.

6. **Recepção de Dados do Servidor**

   - `data = s.recv(1024)` Recebe até 1024 bytes de dados enviados pelo servidor e armazena-os na variável `data`.

7. **Exibição da Mensagem Recebida**

   - `print(f'Recebido do servidor: data.decode()')` Decodifica os dados recebidos do servidor (convertendo de bytes para string) e imprime a mensagem.

Para checar se a porta está no ar, caso precisa: 'netstat -tuln | grep 65432'

7.3.3 Execução

Para testar o código,

1. **Execute o Código do Servidor**

   - Abra um terminal.
   - Navegue até o diretório onde está o arquivo do servidor.
   - Execute o código do servidor com o comando apropriado (por exemplo, `python server.py`).
   - O servidor começará a escutar por conexões.

2. **Execute o Código do Cliente**

   - Abra um segundo terminal.
   - Navegue até o diretório onde está o arquivo do cliente.

- Execute o código do cliente com o comando apropriado (por exemplo, `python client.py`).
- O cliente enviará uma mensagem para o servidor e receberá uma resposta.

O servidor deve receber a mensagem do cliente e enviá-la de volta. O cliente então exibirá a mensagem recebida do servidor.

7.3.4 Integração com Flask

O Flask é ideal para lidar com requisições HTTP, o código fica bem menos verboso.

```
1  from flask import Flask, request, jsonify
2  import socket
3
4  app = Flask(__name__)
5
6  @app.route('/send', methods=['POST'])
7  def send_message():
8      message = request.form['message']
9
10     HOST = '127.0.0.1'
11     PORT = 65432
12     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s
           :
13         s.connect((HOST, PORT))
14         s.sendall(message.encode())
15         response = s.recv(1024)
16
17     return jsonify({'response': response.decode()})
18
19 if __name__ == '__main__':
20     app.run(port=5000)
```

8 Camada de Aplicação

É a camada mais próxima do usuário e fornece os protocolos e serviços que as aplicações utilizam para se comunicar através da rede. Exemplos de protocolos nessa camada incluem HTTP, FTP, SMTP, DNS, entre outros (Forouzan e Fegan 2010).

Material: Implementações feitas em Python.

8.1 Desenvolvimento de um servidor web simples com Python (Flask ou Django)

Este documento descreve o funcionamento de um código Python que implementa um servidor web simples. O servidor escuta conexões na porta 8000 de `localhost` (127.0.0.1) e processa requisições HTTP, retornando o conteúdo de arquivos solicitados, como HTML ou arquivos de texto. Caso o arquivo solicitado não seja encontrado, o servidor retorna uma página de erro 404.

8.1.1 Configuração Inicial do Servidor

8.1.2 Versão sem Flask

Primeiro, o servidor é configurado para escutar em um endereço e porta específicos:

```
myHost = '127.0.0.1'
myPortNumber = 8000

# Configurando o servidor
server = socket(AF_INET, SOCK_STREAM)
server.bind((myHost, myPortNumber))
```

```
server.listen(1)
```

- `myHost` e `myPortNumber` definem o endereço IP e a porta em que o servidor vai escutar.
- O objeto `server` é um socket que usa o protocolo `AF_INET` (IPv4) e `SOCK_STREAM` (TCP), representando um servidor TCP.
- A função `bind` associa o socket ao endereço IP e porta especificados.
- A função `listen(1)` faz o servidor escutar até uma conexão de cliente por vez.

8.1.3 Aceitando Conexões

O servidor entra em um loop onde aceita conexões e processa requisições:

```
while True:
    connection, address = server.accept()
    print('Servidor conectado por', address)

    request = connection.recv(1024).decode('utf-8')
    print("Requisição recebida:", request)
```

- `accept()` espera por uma conexão de um cliente e retorna um novo socket `connection` e o endereço do cliente `address`.
- `recv(1024)` recebe até 1024 bytes de dados da conexão. O conteúdo recebido é decodificado de bytes para uma string usando `utf-8`.

8.1.4 Processamento da Requisição

O servidor extrai o nome do arquivo solicitado e tenta abri-lo:

```
messageTemp = request.split(' ')
if len(messageTemp) > 1:
    messageFile = messageTemp[1][1:]

    if messageFile == '':
        messageFile = 'index.html'
```

- A requisição HTTP é dividida em partes usando `split(' ')`. A segunda parte da requisição, `messageTemp[1]`, contém o caminho do arquivo solicitado.
- `messageFile = messageTemp[1][1:]` remove a barra inicial `/` para obter o nome do arquivo.
- Se o caminho estiver vazio, o servidor assume que o cliente solicitou o arquivo `index.html`.

8.1.5 Resposta ao Cliente

O servidor tenta abrir e ler o arquivo solicitado:

```
try:
    with open(messageFile, 'rb') as file:
        response = file.read()

    if messageFile.endswith('.html'):
        tipoarquivo = 'text/html'
    elif messageFile.endswith('.txt'):
        tipoarquivo = 'text/plain'
    else:
        tipoarquivo = 'application/octet-stream'

    header = 'HTTP/1.1 200 OK\n'
    header += 'Content-Type: ' + tipoarquivo + '\n\n'
```

- `open(messageFile, 'rb')` abre o arquivo em modo de leitura binária. Se o arquivo for encontrado, o conteúdo é lido e armazenado em `response`.
- O tipo de conteúdo (`Content-Type`) é determinado pela extensão do arquivo:
  - `.html`: `text/html`
  - `.txt`: `text/plain`
  - Outros: `application/octet-stream` (tipo genérico)
- O servidor constrói um cabeçalho HTTP com o status `200 OK` e o tipo de conteúdo correspondente.

Se o arquivo não for encontrado, o servidor retorna um erro 404:

```
except Exception as e:
    header = 'HTTP/1.1 404 Not Found\n\n'
    response = '<html><meta charset="utf-8"><body><center><h3>Erro 404: Arquivo não e
               '<p>Servidor Python</p></center></body></html>'.encode('utf-8')
```

8.1.6 Envio da Resposta e Fechamento da Conexão

Finalmente, o servidor envia a resposta HTTP ao cliente e fecha a conexão:

```
respostafinal = header.encode('utf-8')
respostafinal += response
connection.send(respostafinal)
connection.close()
```

- O cabeçalho e o conteúdo da resposta são concatenados e codificados em bytes.
- `connection.send(respostafinal)` envia a resposta ao cliente.
- `connection.close()` fecha a conexão com o cliente.

8.1.7 Tratamento de Interrupção e Liberação de Recursos

O código também inclui tratamento para encerrar o servidor com segurança quando o usuário interrompe a execução (por exemplo, pressionando `Ctrl+C`):

```
except KeyboardInterrupt:
    print("\nServidor encerrado pelo usuário.")

finally:
    server.close()
    print(f"Porta {myPortNumber} liberada. Servidor fechado.")
```

- `KeyboardInterrupt` captura a interrupção do teclado e permite que o servidor seja encerrado com segurança.
- `finally` garante que o socket do servidor seja fechado e a porta liberada, evitando que a porta fique ocupada após o término do programa.

8.1.8 Versão com Flask

O seguinte código Python implementa um servidor web básico usando o framework Flask:

```
1  from flask import Flask, send_from_directory, abort
2
3  app = Flask(__name__)
4
5  @app.route('/')
6  def serve_index():
7      return send_from_directory('.', 'index.html')
8
9  @app.route('/<path:filename>')
10 def serve_file(filename):
11     try:
12         return send_from_directory('.', filename)
```

```
13      except Exception as e:
14          abort(404, description="Arquivo n o encontrado")
15
16  if __name__ == '__main__':
17      app.run(host='127.0.0.1', port=8000)
```

8.1.9 Importações

- **Flask**: Framework que facilita a criação de aplicações web.
- **send_from_directory**: Função que serve arquivos a partir de um diretório específico.
- **abort**: Função que gera uma resposta HTTP de erro, como o 404.

8.1.10 Rotas

- `@app.route('/')`: Define a rota para a raiz (`/`). Serve o arquivo `index.html` do diretório atual.
- `@app.route('/<path:filename>')`: Define uma rota para qualquer caminho (`/<filename>`). Serve o arquivo solicitado ou retorna um erro 404 se o arquivo não for encontrado.

8.1.11 Execução do Servidor

O servidor é iniciado na `127.0.0.1` na porta `8000` com o comando `app.run()`.

8.1.12 Vantagens de Usar Flask

- Simplifica a manipulação de rotas e respostas HTTP.
- Facilita o gerenciamento de arquivos estáticos.

- O tratamento de erros e exceções é mais robusto.

Com Flask, o código é mais limpo e fácil de manter, e o framework cuida de muitos detalhes que precisam ser gerenciados manualmente em um servidor socket puro.

8.2 Captura e análise de tráfego HTTP com Wireshark

8.2.1 O que é Wireshark?

Wireshark é uma ferramenta de análise de redes de código aberto amplamente utilizada para capturar e inspecionar o tráfego de rede em tempo real. É uma ferramenta crucial para administradores de rede, engenheiros e profissionais de segurança para resolver problemas de rede, monitorar o tráfego e diagnosticar questões.

8.2.2 Para que serve Wireshark?

- **Captura de Pacotes**: Wireshark captura pacotes de dados transmitidos em uma rede. Pode capturar pacotes em tempo real ou abrir arquivos de captura previamente gravados.
- **Análise de Tráfego**: Permite examinar o conteúdo dos pacotes e identificar padrões de tráfego, protocolos utilizados e comportamento de aplicações.
- **Diagnóstico de Problemas**: Ajuda na identificação de problemas de rede, como latência, perda de pacotes e outros problemas de conectividade.
- **Segurança de Rede**: Utilizado para identificar tráfego suspeito e potenciais ameaças de segurança, como ataques de malware, tentativas de invasão e vazamentos de dados.
- **Monitoramento de Performance**: Permite monitorar o desempenho da rede e avaliar a eficiência dos protocolos e serviços em uso.
- **Educação e Treinamento**: Usado para fins educacionais para ensinar sobre redes e protocolos, oferecendo uma visão detalhada de como as redes funcionam.

8.2.3 Como Funciona?

Wireshark captura pacotes usando um driver de captura que intercepta o tráfego de rede na interface selecionada. Depois, ele decodifica e exibe o conteúdo

dos pacotes em uma interface gráfica, permitindo a análise detalhada dos dados.

8.2.4 Recursos Principais

- **Interface Gráfica Intuitiva**: Facilita a visualização e análise dos pacotes capturados.
- **Filtragem e Pesquisa**: Oferece ferramentas para filtrar e pesquisar pacotes específicos.
- **Decodificação de Protocolos**: Suporta uma vasta gama de protocolos e fornece uma interpretação detalhada dos dados.

Wireshark é uma ferramenta essencial para a análise de redes e segurança, fornecendo uma visão profunda do tráfego de dados e ajudando a resolver diversos problemas relacionados à rede.

8.2.5 Como as portas de Rede funcionam?

8.2.6 Introdução

As portas de rede são um conceito fundamental em redes de computadores que permitem a comunicação entre diferentes dispositivos e aplicações através de uma única interface de rede. Elas desempenham um papel crucial na multiplexação e no direcionamento do tráfego de rede.

8.2.7 O que são Portas de Rede?

Portas de rede são identificadores numéricos usados pelos protocolos de comunicação para distinguir entre diferentes serviços e aplicações que estão rodando em um mesmo dispositivo. Cada porta é associada a um número de 16 bits, resultando em um intervalo de valores de 0 a 65535.

8.2.8 Como Funcionam as Portas de Rede?

Quando um dispositivo deseja enviar dados para outro dispositivo na rede, ele utiliza um número de porta para indicar qual serviço ou aplicação deve receber os dados. O processo pode ser descrito em várias etapas:

1. **Estabelecimento da Conexão**: O dispositivo cliente inicia uma conexão com um servidor especificando um número de porta. O servidor está escutando em uma porta específica para aceitar conexões.

2. **Endereçamento**: A comunicação é direcionada ao endereço IP do servidor e ao número da porta. O endereço IP identifica o dispositivo na rede, enquanto a porta identifica o serviço específico no dispositivo.

3. **Multiplexação**: O sistema operacional do servidor usa o número da porta para encaminhar os dados para o serviço ou aplicação apropriada. Isso permite que múltiplos serviços funcionem simultaneamente em um único dispositivo.

4. **Resposta e Comunicação**: Após o recebimento dos dados, o serviço/processo responde através da mesma porta utilizada para a conexão. O tráfego de retorno também usa o número da porta para garantir que os dados sejam entregues ao processo correto.

8.2.9 Tipos de Portas

Existem três principais tipos de portas:

- **Portas Bem Conhecidas (0 a 1023)**: São usadas por protocolos comuns e serviços conhecidos, como HTTP (porta 80), HTTPS (porta 443) e FTP (porta 21).

- **Portas Registradas (1024 a 49151)**: São usadas por aplicações de usuário e serviços que não são tão amplamente conhecidos. Os números de porta nesta faixa podem ser registrados para evitar conflitos.

- **Portas Dinâmicas ou Efêmeras (49152 a 65535)**: São usadas temporariamente por clientes ao estabelecer conexões com servidores. Estas portas são alocadas dinamicamente e utilizadas apenas enquanto a conexão está ativa.

8.2.10 Definição de Sockets

Sockets são interfaces de comunicação que permitem a troca de dados entre processos, seja na mesma máquina ou em máquinas diferentes em uma rede. Eles fornecem um meio para enviar e receber dados através de redes, como a Internet ou uma rede local.

8.2.11 Funcionamento dos Sockets

Um socket é uma combinação de um endereço IP e um número de porta, que identifica um ponto final de comunicação em uma rede. O funcionamento dos sockets pode ser descrito através das seguintes etapas:

1. **Criação do Socket**: Um socket é criado por um processo usando uma chamada de sistema. Esse processo define o tipo de socket, como TCP ou UDP, e as opções de configuração necessárias.

2. **Binding (Vinculação)**: Para sockets de servidor, o socket é associado a um endereço IP e um número de porta específicos. Esse processo é conhecido como ”binding”.

3. **Escuta (Listen)**: Um socket de servidor começa a escutar conexões em uma porta específica. O processo fica aguardando por conexões de clientes.

4. **Conexão e Comunicação**: Quando um cliente deseja se conectar, ele cria um socket e solicita uma conexão com o servidor. O servidor aceita a conexão e estabelece um canal de comunicação. Uma vez estabelecida a conexão, dados podem ser enviados e recebidos através dos sockets.

5. **Encerramento da Conexão**: Após a comunicação ser concluída, a conexão é encerrada e os sockets são fechados.

8.2.12 Tipos de Sockets

Os principais tipos de sockets são:

- **Sockets TCP**: Utilizam o protocolo Transmission Control Protocol (TCP) para comunicação. São orientados à conexão e garantem que os dados sejam entregues de forma confiável e na ordem correta.
- **Sockets UDP**: Utilizam o protocolo User Datagram Protocol (UDP). São orientados a datagramas e não garantem a entrega dos dados, proporcionando uma comunicação mais rápida, porém sem garantia de entrega ou ordem.

8.2.13 Exemplo de Uso

Um exemplo típico de uso de sockets é uma aplicação cliente-servidor onde o servidor escuta em uma porta específica e o cliente se conecta a essa porta para enviar ou receber dados. Esse modelo é amplamente utilizado em aplicações de rede, como servidores web, clientes de e-mail e jogos online.

8.2.14 Análise HTTP de uma página WEB

Para a análise HTTP, foi utilizado o wireshark, software de captura de pacotes de interfaces de rede. Já a página WEB que foi utilizada para os testes foi a http://www.bancocn.com/, visto que é HTTP e é um site feito para testes, pela equipe Solyd Tecnologia, empresa de treinamento de segurança da informação.

Primmeiramente, ao acessar o wireshrk, é possível ver quais interfaces o usuário pode rastrear.

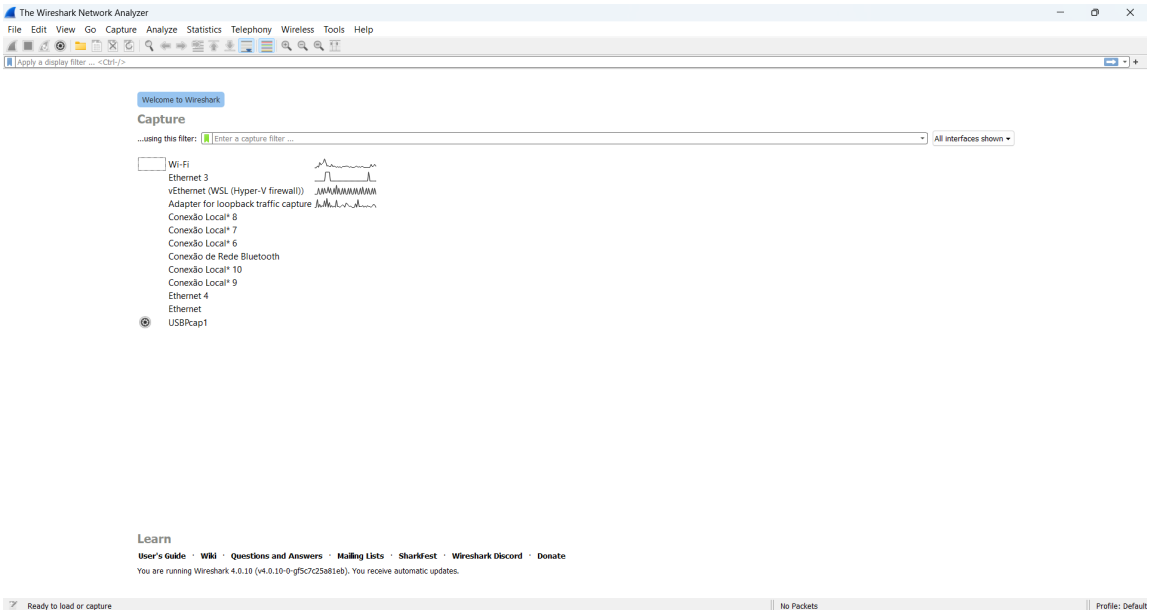


Figura 13 – Página inicial do wireshark. Fonte: próprio autor.

Foi escolhido o Wi-Fi, ou seja, é rastreado as conexões que partem do meu servidor de internet.

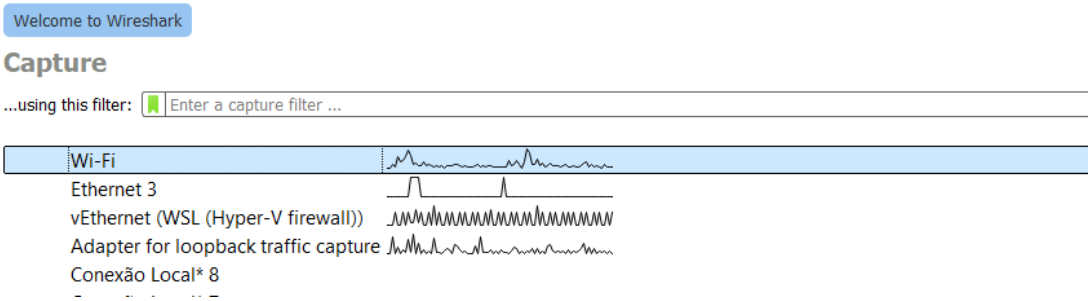


Figura 14 – Escolha do wifi. Fonte: próprio autor.

Na sequência, foi digitado http na barra dos filtros, para que pudesse rastrear somente as requisições HTTP, das informações que chegam ao Wireshark, dos meus acessos.

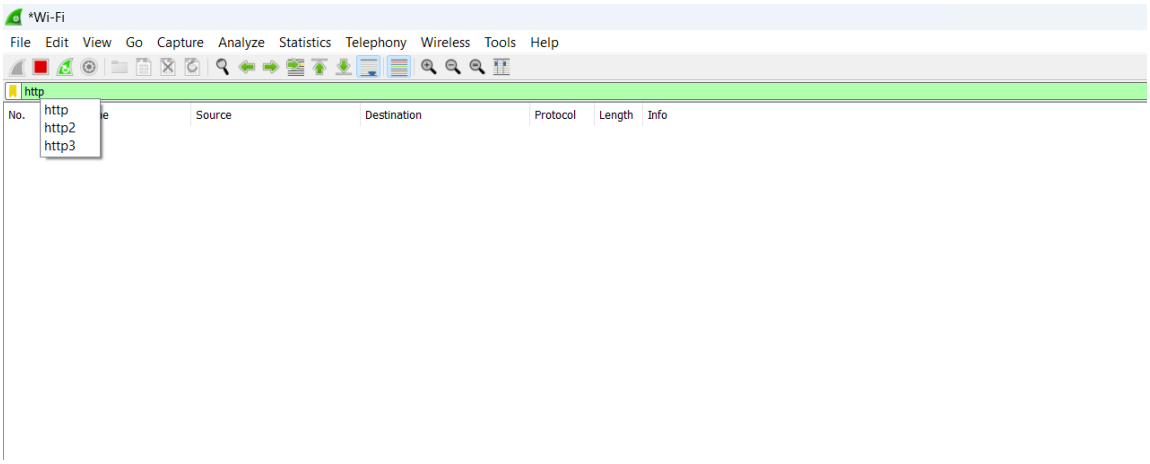


Figura 15 – Escolha do http como filtro do wireshark. Fonte: próprio autor.

A imagem do site que foi acessado.

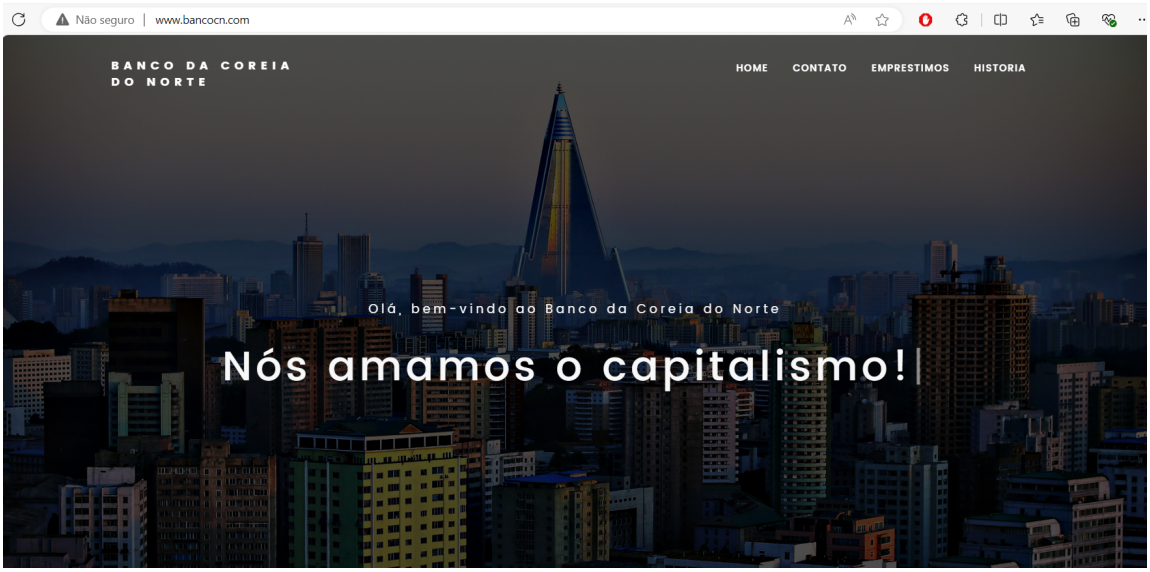


Figura 16 – Site desprotejido. Fonte: http://www.bancocn.com/.

Aqui é possível observar as requisições feitas e os pacotes gerados.

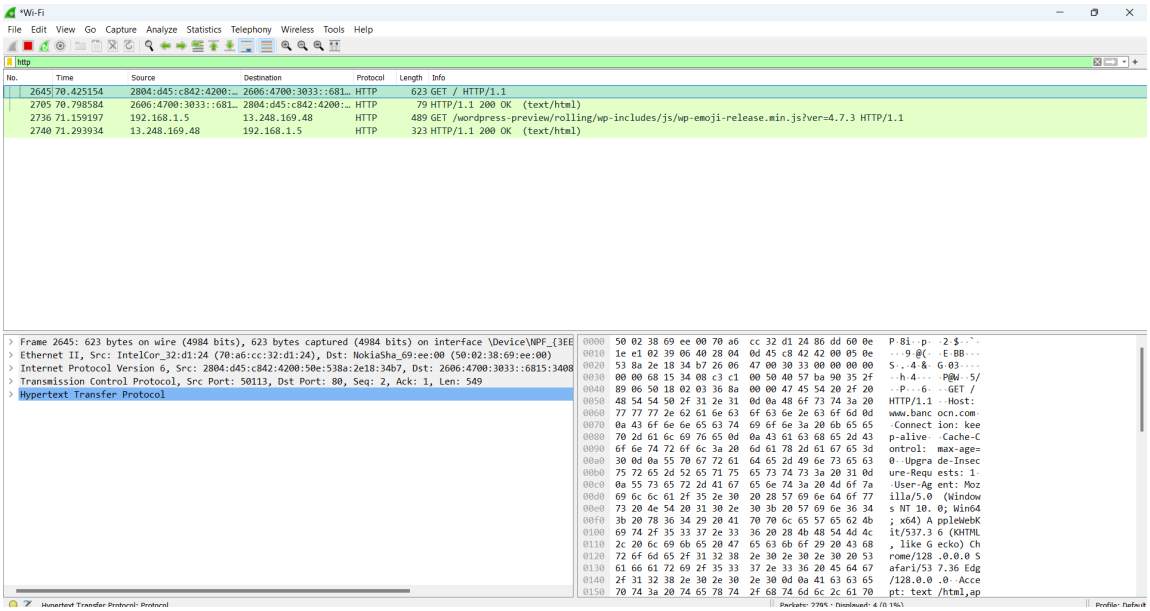


Figura 17 – Página inicial do wireshark. Fonte: próprio autor.

Por fim, é possível coletar as informações de qual software de busca estou acessando a página, qual a página, qual o motor de busca, qual o idioma da página e quais os tipos de dados que estão rpesentes na página.

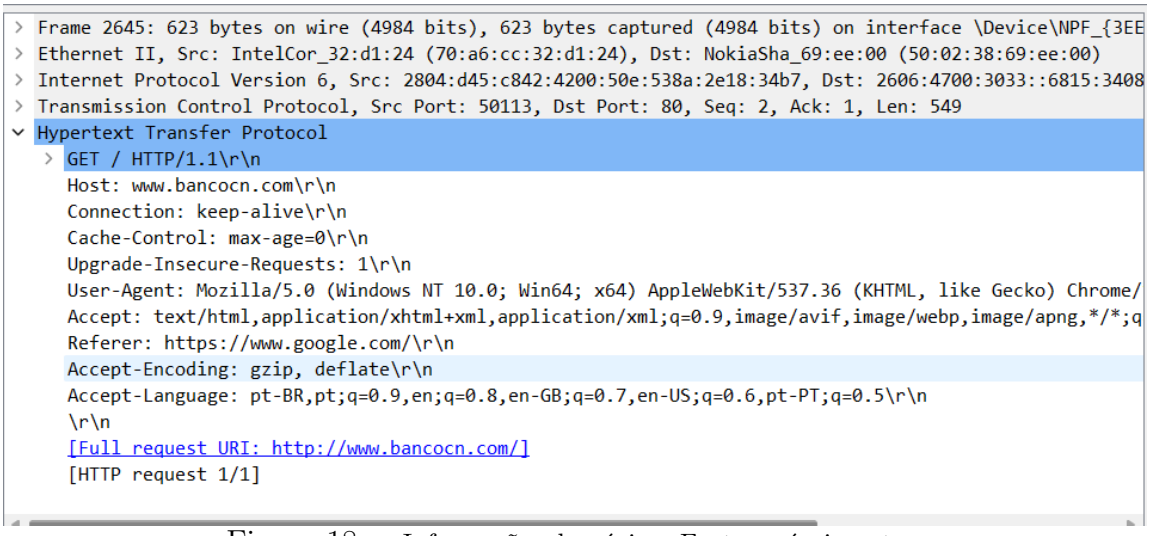


Figura 18 – Informações da página. Fonte: próprio autor.

Vale ressaltar que o Wireshark é uma ferramenta poderosa de rastrear informações. Se a página tivesse um login, por exemplo, ele seria capaz de capturar a senha que o usuário está colocando na página. Ou seja. Conexões que são HTTP, são extremamente perigosas, o recomendado é utilizar as conexões HTTPS, que são protocolos criptografados, ou seja, é possível capturar a informação também, mas esta vai chegar criptografada, de forma que o interseptante não irá conseguir identificar o que está sendo informado naquela mensagem.

8.2.15 HTTP e HTTPS

O protocolo de transferência de hipertexto (HTTP) é um protocolo ou conjunto de regras de comunicação para comunicação entre cliente e servidor. Quando você visita um site, o navegador envia uma solicitação HTTP ao servidor Web, que responde com uma resposta HTTP. O servidor Web e o navegador trocam dados como texto simples. Resumindo, o protocolo HTTP é a tecnologia subjacente que alimenta a comunicação de rede. Como o nome sugere, o protocolo de transferência de hipertexto seguro (HTTPS) é uma versão mais segura ou uma extensão do HTTP. No HTTPS, o navegador e o servidor estabelecem uma conexão segura e criptografada antes de transferir dados (O que é a computação em nuvem?).

8.2.16 Como o protocolo HTTP funciona?

O HTTP é um protocolo de camada de aplicação no modelo de comunicação de rede Open Systems Interconnection (OSI). Ele define vários tipos de solicitações e respostas. Por exemplo, quando deseja visualizar alguns dados de um site, você envia a solicitação HTTP GET. Se quiser enviar algumas informações, como preencher um formulário de contato, você envia a solicitação HTTP PUT (O que é a computação em nuvem?).

Da mesma forma, o servidor envia diferentes tipos de respostas HTTP na forma de códigos numéricos e dados. Veja alguns exemplos:

8.2.17 200 - OK

O código **200 - OK** indica que a solicitação foi bem-sucedida e o servidor retornou a resposta esperada. Esse é o código de status padrão para solicitações HTTP bem-sucedidas.

8.2.18 400 - Solicitação Inválida

O código **400 - Solicitação Inválida** indica que a solicitação do cliente é inválida ou malformada. Isso pode ocorrer se a sintaxe da solicitação estiver incorreta ou se faltarem parâmetros necessários.

8.2.19 404 - Recurso Não Encontrado

O código **404 - Recurso Não Encontrado** indica que o recurso solicitado pelo cliente não pôde ser encontrado no servidor. Isso pode ocorrer se o URL estiver incorreto ou se o recurso foi removido.

Essa comunicação de solicitação/resposta geralmente é invisível para os usuários. É o método de comunicação que o navegador e os servidores Web usam, então a World Wide Web funciona de forma consistente para todos (O que é a computação em nuvem?).

8.2.20 Como o protocolo HTTPS funciona?

O HTTP transmite dados não criptografados, ou seja, as informações enviadas de um navegador podem ser interceptadas e lidas por terceiros. Esse não era um processo ideal, então foi estendido para o HTTPS, para adicionar outra camada de segurança à comunicação. O HTTPS combina solicitações e respostas HTTP com a tecnologia SSL e TLS (O que é a computação em nuvem?).

Os sites HTTPS devem obter um certificado SSL/TLS de uma autoridade de certificação (CA) independente. Esses sites compartilham o certificado com o navegador antes de trocar dados para estabelecer confiança. O certificado SSL também

contém informações criptográficas, para que o servidor e os navegadores da Web possam trocar dados criptografados ou codificados. O processo funciona assim:

Você visita um site HTTPS digitando o formato de URL https://na barra de endereço do seu navegador. O navegador tenta verificar a autenticidade do site solicitando o certificado SSL do servidor. O servidor envia o certificado SSL que contém uma chave pública como resposta. O certificado SSL do site comprova a identidade do servidor. Quando o navegador estiver satisfeito, ele usará a chave pública para criptografar e enviar uma mensagem que contém uma chave de sessão secreta. O servidor web usa sua chave privada para descriptografar a mensagem e recuperar a chave de sessão. Em seguida, ele criptografa a chave da sessão e envia uma mensagem de confirmação ao navegador. Agora, o navegador e o servidor da Web mudam para usar a mesma chave de sessão para trocar mensagens com segurança (O que é a computação em nuvem?).

8.2.21 Qual é a diferença entre HTTP/2, HTTP/3 e HTTPS?

A versão HTTP original lançada em 1996-97 se chamava HTTP/1.1. O HTTP/2 e o HTTP/3 são versões atualizadas do próprio protocolo. O sistema de transferência de dados foi modificado para torná-lo mais eficiente. Por exemplo, o HTTP/2 troca dados em formato binário em vez de textual. Ele também permite que os servidores transmitam respostas proativamente aos caches dos clientes em vez de esperarem por uma nova solicitação HTTP. O HTTP/3 é relativamente novo, mas tenta levar o HTTP/2 um passo adiante. O objetivo do HTTP/3 é oferecer suporte a transmissões em tempo real e a outros requisitos modernos de transferência de dados com mais eficiência (O que é a computação em nuvem?).

O HTTPS prioriza as preocupações com a segurança dos dados no HTTP. Os sistemas modernos usam HTTP/2 com SSL/TLS como HTTPS. À medida que o HTTP/3 amadurece, as tecnologias de navegador e servidor também o integrarão ao HTTPS.

8.3 estudo de protocolos de aplicação como FTP, DNS, SMTP

8.3.1 Introdução

Protocolos de aplicação são fundamentais para a comunicação entre sistemas na rede. Eles definem regras e convenções para a troca de dados entre aplicações. Neste estudo, exploraremos três protocolos de aplicação amplamente utilizados: FTP, DNS e SMTP.

8.3.2 File Transfer Protocol (FTP)

8.3.3 O que é FTP?

O File Transfer Protocol (FTP) é um protocolo de rede utilizado para a transferência de arquivos entre um cliente e um servidor. O FTP opera na camada de aplicação do modelo OSI e usa a arquitetura cliente-servidor para enviar e receber arquivos.

8.3.4 Funcionamento

O FTP utiliza duas portas principais para comunicação:

- **Porta 21**: Usada para comandos de controle. O cliente envia comandos ao servidor através desta porta.
- **Porta 20**: Usada para a transferência de dados, quando o modo de transferência ativo é utilizado.

Os principais comandos FTP incluem `USER` (para autenticação), `PASS` (para senha), `LIST` (para listar arquivos), `RETR` (para recuperar arquivos) e `STOR` (para enviar arquivos).

8.3.5 Exemplo de Comando FTP

Para conectar a um servidor FTP e listar arquivos, o cliente pode usar os seguintes comandos:

```
ftp ftp.example.com
USER username
PASS password
LIST
```

8.3.6 Domain Name System (DNS)

8.3.7 O que é DNS?

O Domain Name System (DNS) é um protocolo de aplicação que traduz nomes de domínio legíveis por humanos em endereços IP numéricos. Ele permite que os usuários acessem recursos na web usando nomes de domínio, como `www.example.com`, em vez de endereços IP.

8.3.8 Funcionamento

O DNS opera de forma hierárquica e distribuída, consistindo em uma rede de servidores DNS. Os principais tipos de registros DNS incluem:

- **A Record**: Mapeia um nome de domínio para um endereço IPv4.
- **AAAA Record**: Mapeia um nome de domínio para um endereço IPv6.
- **CNAME Record**: Define um nome de domínio como um alias para outro nome de domínio.
- **MX Record**: Define os servidores de e-mail para o domínio.

Quando um cliente precisa resolver um nome de domínio, ele envia uma consulta DNS para um servidor DNS recursivo, que então encaminha a consulta para servidores DNS autoritativos se necessário.

8.3.9 Exemplo de Consulta DNS

Para verificar o endereço IP associado a um domínio, o cliente pode usar o comando `nslookup`:

```
nslookup www.example.com
```

8.3.10 Simple Mail Transfer Protocol (SMTP)

8.3.11 O que é SMTP?

O Simple Mail Transfer Protocol (SMTP) é um protocolo de comunicação utilizado para enviar e encaminhar e-mails entre servidores de e-mail. O SMTP opera na camada de aplicação e usa a porta 25 para a troca de mensagens de e-mail.

8.3.12 Funcionamento

O SMTP é um protocolo baseado em texto que usa comandos e respostas para enviar e-mails. Os principais comandos SMTP incluem `HELO` (para iniciar uma sessão), `MAIL FROM` (para especificar o remetente), `RCPT TO` (para especificar o destinatário) e `DATA` (para enviar o corpo da mensagem).

8.3.13 Exemplo de Envio de E-mail

Para enviar um e-mail usando SMTP, o cliente pode usar os seguintes comandos:

```
HELO mail.example.com
MAIL FROM:<sender@example.com>
```

```
RCPT TO:<recipient@example.com>
DATA
Subject: Teste
Olá, este é um teste.
.
QUIT
```

8.3.14 Conclusão

Os protocolos de aplicação FTP, DNS e SMTP desempenham papéis essenciais na comunicação de rede. O FTP é utilizado para a transferência de arquivos, o DNS para a resolução de nomes de domínio e o SMTP para o envio de e-mails. Compreender esses protocolos é fundamental para o gerenciamento e a operação eficiente de redes e serviços de comunicação.

Referências

ANDRADE, R.; KAMIENSKI, C.; SOUSA, D.; SADOK, D. O algoritmo sqm-response para controle de congestionamento do protocolo tcp. In: *Anais do 21º Simpósio Brasileiro de Redes de Computadores (SBRC-2003)*. Natal-RN, Brasil: [s.n.], 2003. Citado na página 87.

FOROUZAN, B. A.; FEGAN, S. C. *Comunicação de Dados e Redes de Computadores*. 4. ed. Porto Alegre: AMGH Editora Ltda, 2010. Tradução: Ariovaldo Griesi, Revisão Técnica: Jonas Santiago de Oliveira. ISBN 978-85-63308-47-4. Citado 10 vezes nas páginas 8, 9, 12, 18, 38, 45, 59, 71, 74 e 97.

KUROSE, J. F.; ROSS, K. W. *Redes de Computadores e a Internet: Uma Abordagem Top-Down*. 6. ed. [S.l.]: Pearson, 2013. ISBN 978-85-8143-677-7. Citado na página 8.

MAIA, L. P. *Arquitetura de Redes de Computadores*. 2. ed. Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora Ltda, 2013. ISBN 978-85-216-2435-6. Citado 3 vezes nas páginas 8, 31 e 51.

NEVES, G. Tcp e udp: Protocolos de transporte nas redes de computadores. 2024. Acessado em: 30 de agosto de 2024. Disponível em: <https://www.linkedin.com/pulse/tcp-vs-udp-gabriel-neves-eguqf/>. Citado na página 74.

O que é a computação em nuvem? <https://www.aws.com>. Acessado em: 30 de agosto de 2024. Citado 3 vezes nas páginas 112, 113 e 114.

O que é a computação em nuvem? 2024. <https://aws.amazon.com>. Acessado em: 30 de agosto de 2024. Citado na página 62.

PETERSON, L. L.; DAVIE, B. S. *Redes de computadores: uma abordagem de sistemas*. Rio de Janeiro, Brasil: Elsevier Editora Ltda., 2013. ISBN 978-85-352-4897-5. Citado 3 vezes nas páginas 8, 18 e 25.

TANENBAUM, A. S. *Redes de Computadores*. 4. ed. Amsterdam, Holanda: Editora Campus, 2003. Citado 2 vezes nas páginas 8 e 12.