Universidade de Brasília

Departamento de Ciência da Computação
Teleinformática e Redes 1

Trabalho 2

Camada Física

Autor:

Lucca Beserra Huguet 16/0013259

Simulador:

Linguagem rust com a biblioteca Plotters

Brasília
22 de setembro de 2021

Conteúdo

1 Introdução

1.1 Descrição do problema

Simular o funcionamento do enlace físico por meio da implementação das seguintes codificações:

- **Binária**
- **Manchester**
- **Bipolar**

1.2 Visão Geral do Simulador

O simulador recebe um input do usuário (string).

Cada caracter dessa string é trasformado em um inteiro de 8 de bits sem sinal, ou seja, um byte. É criado então um vetor, onde cada elemento é um desses bytes.

Estes bytes são codificados, transmitidos pelo "meio de comunicação", depois decodificados e transformados novamente em strings.

Durante esse processo são geradas as simulações, que ficam armazenadas em gifs.

Estes gifs mudam byte a byte, mostrando a sequência de bits existente. Foram simulados os sinais após o processamento da Camada de Aplicação Transmissora, depois do processamento da Camada Física Transmissora e depois do processamento da Camada Física Receptora. Naturalmente, a primeira e a terceira simulações mostrarão os mesmo bytes o que assegura a correção do algoritmo.

2 Implementação

Descrição detalhada do desenvolvimento com diagramas ilustrativos, o funcionamento dos protocolos, procedimentos utilizados, bem como decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado.

2.1 Aplicação Transmissora

Na aplicação transmissora recebemos uma string digitada pelo usuário. Esta é salva numa variável mutável e transmitida para a camada de aplicação transmissora. A variável é mutável, pois a função de ler inputs do usuário receve este tipo de variáveis.
(Obs: mutabilidade é um conceito de rust que tem como objetivo deixar o seu código mais seguro)

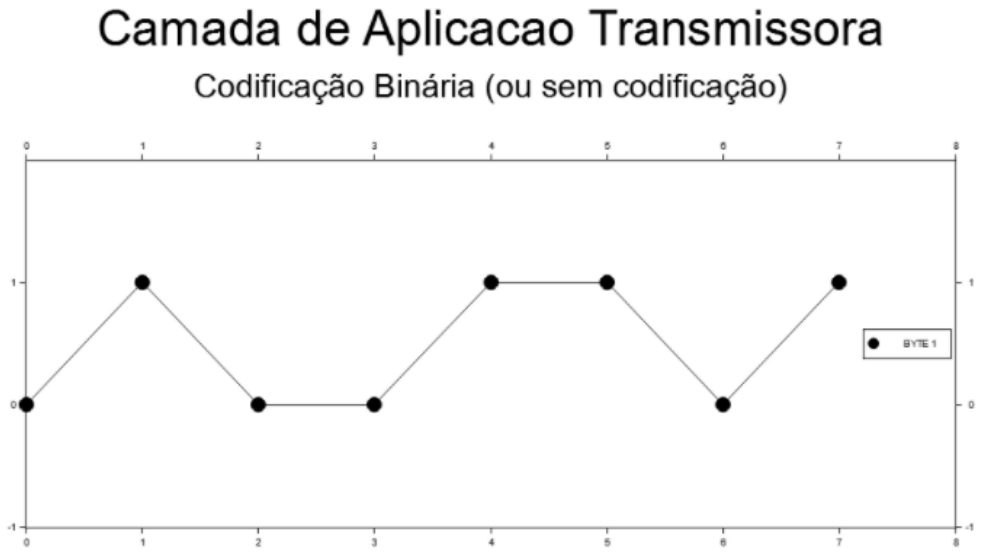
2.2 Camada de Aplicação Transmissora

Na camada de aplicação transmissora recebemos a string obtida na aplicação transmissora. Por meio de um for, inserimos cada char, interpretado como u8, no recém criado vetor (de tipo Vec $< u8 >$ e mutável). O tipo u8 trata-se de um inteiro de 8 bits, o que podemos considerar também como um byte. Com o byte em mãos, podemos transmitir o sinal usando pouca memória e fazer operações eficientes em cima de cada bit.

"MIMO", por exemplo, se transforma em:
- **M = 0100 1101**
- **I = 0100 1001**
- **M = 0100 1101**
- **O = 0100 1111**

No algoritmo, um plot deste sinal é salvo em
"plotters-doc-data \Camada_de_Aplicacao_Transmissora.gif"

Um dos bytes descritos no gif, por exemplo, é o byte 1, que representa "M". Veja abaixo.



Note que o algoritmo não funciona para caracteres non-ASCII.

2.3 Camada Física Transmissora

Nesta camada efetuaremos as codificações em cima dos bits que temos. Note que eles já estão na codificação binária, e portanto não são necessárias operações para tal. Já para as codificações manchester e bipolar precisaremos dobrar o número de bits, pois precisamos representar subida e decida no caso da Manchester, e -1, 0 e 1 no caso da bipolar.
Após a codificação o sinal é enviado para a função meio_de_comunicação(). No algoritmo, um plot deste sinal é salvo em "plotters-doc-data \Camada_Fisica_Transmissora.gif"

2.4 Codificação Manchester

Nesse codificão temos que o bit 0 é representado por uma subida, e o bit 1 por uma descida. Representaremos subida por 01, e descida por 10.

Continuando ainda com o nosso exemplo, "MIMO", temos:
- **M = 0100 1101 = 0110 0101 — 1010 0110**
- **I = 0100 1001 = 0110 0101 — 1001 0110**

- M = 0100 1101 = 0110 0101 — 1010 0110
- O = 0100 1111 = 0110 0101 — 1010 1010

2.5 Codificação Bipolar

Nesse codificão temos que o bit 0 é representado por 0, e o bit 1 pode ser 1 ou -1, dependendo de quantos 1's foram encontrados no byte. Se um número par de 1's for encontrado, será 1, senão, será -1. Representaremos 0 por 00, 1 por 01 e -1 por 10.

Continuando ainda com o nosso exemplo, "MIMO", temos:
- M = 0100 1101 = 0001 0000 — 1001 0010
- I = 0100 1001 = 0001 0000 — 1000 0001
- M = 0100 1101 = 0001 0000 — 1001 0010
- O = 0100 1111 = 0001 0000 — 1001 1001

2.6 Meio de comunicação

Neste trecho, o sinal comumente passaria por sua longa comuta até a camada física receptora, exposto a todo tipo de ruído, porém, nesta simulação, não houve representação do ruído, o sinal simplesmente segue, inalterado, até a camada física de recepção

2.7 Camada Física Receptora

Recebemos o sinal do meio de comunicação, que agora será decodificado para seus bytes originais. Para tal usamos funções específicas que irão decodificar o sinal de acordo com a codificação usada anteriormente.

Assim, obtemos novamente o sinal de bits na simples codificação binária:
- 0100 1101
- 0100 1001
- 0100 1101
- 0100 1111

2.8 Camada de Aplicação Receptora

Agora que temos os bytes originais, podemos efetuar a recuperação dos caracteres, formando novamente a string inicial. Para tal, usamos a função str::from_utf8 para obter o tipo str, e por fim a função to_string(), que transforma o valor do tipo str para String (rust possui dois tipos de string). Com a string em mãos novamente, a enviamos para a Aplicação Receptora.

Assim, obtemos novamente a string original:
• 0100 1101 = M
• 0100 1001 = I
• 0100 1101 = M
• 0100 1111 = O
= MIMO

No algoritmo, um plot deste sinal é salvo em
"plotters-doc-data \Camada_Fisica_Receptora.gif"

2.9 Aplicação Receptora

Dentro da função da aplicacao_receptora(), obtemos a string já recuperada e por fim printamos na tela a string, que se trata da mesma string inicialmente inserida, dado que houve nenhum problema no processo (durante os testes, o algoritmo não encontrou problemas).

3 Conclusão

Nota-se que é necessário um grande poder computacional para a codificação e decodificação do sinal, ainda mais quando se considera a magnitude da Internet. Também fica clara a grande vulnerabilidade do sinal, que se exposto a um ruído, poderia perder informação ou tê-la alterada, causando problemas que podem ser triviais ou gravissímos, a depender do serviço atingido.

Durante o trabalho, houve também por minha parte a percepção da importância de tratar o sinal nos níveis mais baixos possíveis, otimizando performance e memória.

Sobre as dificuldades encontradas, houve diria que a maior foi entender os mecanismos de rust, visto que nunca havia usado a linguagem antes. Mas como a linguagem se propõe, pude trabalhar num nível baixo, obtendo alta performance (como C e Cpp) mas tendo a vantagem de ter um compilador exigente que protege o programador de vários erros possíveis, em especial relacionados a memória.

Ainda nesse aspecto,nota-se que em rust não é preciso se preocupar com liberação de memória, pois inclusive não há garbage collection, e o programa libera memória automaticamente de acordo com o escopo da variável e sua lifetime (conceito de rust). Como discutido por muitos na comunidade de tecnologias e como pude perceber durante o trabalho, rust é uma ótima alternativa para processos onde performance e segurança são fundamentais, em muitos casos substituindo C ou Cpp para projetos greenfield ou até mesmo portações de código.

Outra dificuldade foi tratar dos bits diretamente, fazendo shifts e bitwise operations em geral, por vezes lidando com 3 loops aninhados.

E ainda, outra parte consideravelmente difícil, foi plotar os bits de forma adequada, entendendo como funciona a biblioteca escolhida para tal.

Finalmente, o trabalho trouxe para mim uma visão consideravelmente mais sólida de como a internet funciona, e que existem importantes tradeoffs presentes seja no desenvolvimento das tecnologias envolvidas, seja na escolha da tecnologia adequada para o seu empreendimento ou uso pessoal.