**UNIVERSIDADE FEDERAL FLUMINENSE**

**INSTITUTO DE COMPUTAÇÃO**

**BACHARELADO EM SISTEMAS DE INFORMAÇÃO**

**TCC00228 - Redes de computadores II para Sistemas de Informação**

**DAVI CHALITA, JOSÉ PAULO DE MELLO GOMES, MATHEUS BALDISSARA**

# TRABALHO SEMESTRAL

**NITERÓI**

**2019**

**Davi Chalita, José Paulo de Mello Gomes, Matheus Baldissara**

# Trabalho semestral

Trabalho apresentado como parte da avaliação semestral da disciplina TCC00228 - Redes de computadores II para Sistemas de Informação do Bacharelado de Sistemas de Informação na Universidade Federal Fluminense - UFF.

Orientador: Prof. Dr. Diego Gimenez Passos

Niterói

2019

# SUMÁRIO


**1 INTRODUÇÃO . . . . . 4**
**1.1 Baixando repositório . . . . . 4**

**2 SEM FOWARD ERROR CORRECTION . . . . . 5**
**2.1 Descrição . . . . . 5**
**2.2 Implementação e execução . . . . . 5**
**2.3 Executando . . . . . 5**
*2.3.0.1 Função codePacket(originalPacket) . . . . . 5*
*2.3.0.2 Função decodePacket(transmittedPacket) . . . . . 5*
*2.3.0.3 Função generateRandomPacket(l) . . . . . 5*
*2.3.0.4 Função geomRand(p) . . . . . 5*
*2.3.0.5 Função insertErrors(codedPacket, errorProb) . . . . . 5*
*2.3.0.6 Função countErrors(originalPacket, decodedPacket) . . . . . 6*

**3 PARIDADE BIDIMENSIONAL . . . . . 7**
**3.1 Descrição . . . . . 7**
**3.2 Implementação e execução . . . . . 7**
**3.3 Executando . . . . . 7**
*3.3.0.1 Função geomRand(p) . . . . . 7*
*3.3.0.2 Função generateRandomPacket(l, linha) . . . . . 7*
*3.3.0.3 Função insertErrors(codedPacket, errorProb) . . . . . 8*
*3.3.0.4 Função countErrors(originalPacket, decodedPacket) . . . . . 8*
*3.3.0.5 Função codePacket(originalPacket,linha,coluna) . . . . . 8*
*3.3.0.6 Função decodePacket(transmittedPacket, linha, coluna) . . . . . 8*
*3.3.0.7 Função somarColunaMatriz(parityMatrix, linha, j) . . . . . 8*
*3.3.0.8 Função somarLinhaMatriz(parityMatrix, coluna, i) . . . . . 8*

**4 CÓDIGO DE HAMMING . . . . . 9**
**4.1 Descrição . . . . . 9**
**4.2 Implementação e execução . . . . . 9**
**4.3 Executando . . . . . 9**
*4.3.0.1 Função geomRand(p) . . . . . 9*
*4.3.0.2 Função insertErrors(codedPacket, errorProb) . . . . . 9*
*4.3.0.3 Função generateRandomPacket(l) . . . . . 9*
*4.3.0.4 Função countErrors(originalPacket, decodedPacket) . . . . . 10*
*4.3.0.5 Função numeroBitsParidade(originalPacket) . . . . . 10*

*4.3.0.6 Função insereEspacosParaBitsParidade(originalPacket) . . . . . 10*
*4.3.0.7 Função hamming(dados) . . . . . 10*
*4.3.0.8 Função hammingCorrecao(codedPacketComErros) . . . . . 10*

**5 ANÁLISE DOS CÓDIGOS . . . . . 11**

**6 BIBLIOGRAFIA . . . . . 13**

# 1 INTRODUÇÃO

Este trabalho visa apresentar, explicar e analisar alguns métodos de detecção e correção de erros utilizados em redes de computadores.

Dentre os diversos métodos existentes, serão abordados três. São eles: sem método de correção, paridade bidimensional e código de Hamming. Nas sessões a seguir, explicaremos sobre os três métodos.

## 1.1 Baixando repositório

```
$ git init
$ git clone https://github.com/DaviChalita/codigos-de-verificacao-de-erros.
$ cd codigos-de-verificacao-de-erros
```

# 2 SEM FOWARD ERROR CORRECTION

## 2.1 Descrição

Aqui, os pacotes não passam por nenhum método de codificação/decodificação. Ou seja, não há a detecção ou correção de erros.

## 2.2 Implementação e execução

Este projeto foi desenvolvido em Python 3.7.0. Portanto, faz-se necessário que esta versão ou uma mais recente esteja instalada.

## 2.3 Executando

```
1 $ py noFEC.py <tam_pacote> <reps> <prob. erro>
```

Onde os parâmetros são, respectivamente: tamanho do pacote, número de repetições e probabilidade de erro.

Veremos, a seguir, uma breve descrição das principais funções utilizadas

#### *2.3.0.1 Função codePacket(originalPacket)*

"Codifica"o pacote. Porém, como não há codificação, apenas retorna o pacote original.

#### *2.3.0.2 Função decodePacket(transmittedPacket)*

"Decodifica"o pacote. Porém, como não decodificação, apenas retorna o pacote transmitido.

#### *2.3.0.3 Função generateRandomPacket(l)*

Gera um pacote aleatório de tamanho l.

#### *2.3.0.4 Função geomRand(p)*

Gera um numero pseudo-aleatorio com distribuicao geometrica.

#### *2.3.0.5 Função insertErrors(codedPacket, errorProb)*

Insere erros no pacote codedPacket com probabilidade errorProb.

#### *2.3.0.6 Função countErrors(originalPacket, decodedPacket)*

Conta a quantidade de erros inseridos, comparando os bits do pacote original com os bits do pacote "decodificado".

# 3 PARIDADE BIDIMENSIONAL

## 3.1 Descrição

Os bits de dados são dispostos em uma matriz m por n e, então, adicionada uma linha e ma coluna para os bits de paridade. Os bits de cada linha e coluna são somados, a paridade deles é calculada e um bit de paridade é adicionada ao final de cada linha e coluna. Casa haja um número par de bits 1, o bit de paridade tem valor 0 (paridade par) ou 1 (paridade ímpar).

Após calculada a paridade, o pacote codificado é enviado para o destinatário. Este, por sua vez, verifica cada célula da matriz de paridade cruzando o bit de paridade da linha com o da coluna, sendo capaz de detectar e corrigir quando há um, e somente um, bit errado.

## 3.2 Implementação e execução

Este projeto foi desenvolvido em Python 3.7.0. Portanto, faz-se necessário que esta versão ou uma mais recente esteja instalada.

## 3.3 Executando

```
1 $ py bidimensional-parity-check.py <tam_pacote> <reps> <prob. erro>
```

Onde os parâmetros são, respectivamente: tamanho do pacote, número de repetições e probabilidade de erro.

Após a escolha dos parâmetros do usuário, os pacotes das matrizes 2x2, 2x3 e 3x3 são gerados e codificados, então os erros são inseridos aleatoriamente. Na decodificação dos pacotes, o programa gera novos bits de paridade sobre os bits do pacote recebido, verificando sua integridade e corrigindo até 1 bit errado por matriz. Então o programa imprime o resultado.

Veremos, a seguir, uma breve descrição das principais funções utilizadas

#### *3.3.0.1 Função geomRand(p)*

Gera um numero pseudo-aleatorio com distribuicao geometrica.

#### *3.3.0.2 Função generateRandomPacket(l, linha)*

Gera um pacote aleatório, passando o tamanho do pacote e a quantidade de linhas da matriz.

#### *3.3.0.3 Função insertErrors(codedPacket, errorProb)*

Insere erros no pacote codedPacket com probabilidade errorProb.

#### *3.3.0.4 Função countErrors(originalPacket, decodedPacket)*

Conta a quantidade de erros inseridos, comparando os bits do pacote original com os bits do pacote "decodificado".

#### *3.3.0.5 Função codePacket(originalPacket,linha,coluna)*

Codifica o pacote, passando o pacote original, a quantidade de linahs e colunas como parâmetros.

#### *3.3.0.6 Função decodePacket(transmittedPacket, linha, coluna)*

Decodifica o pacote transmitido.

#### *3.3.0.7 Função somarColunaMatriz(parityMatrix, linha, j)*

Soma todos os valores na coluna j da parityMatrix. A variável linha é utilizada para percorrer todas as linhas da matriz.

#### *3.3.0.8 Função somarLinhaMatriz(parityMatrix, coluna, i)*

Soma todos os valores na linha i da parityMatrix. A variável coluna é utilizada para percorrer todas as colunas da matriz.

# 4 CÓDIGO DE HAMMING

## 4.1 Descrição

Código de correção de erro (corrige no máximo 1 bit por pacote), seus bits de paridade se encontram nas posições com potência de 2, eles são inseridos durante o processo de codificação.

Valores dos bits de paridade são designados comparando com os bits relacionados a eles(exemplo: paridade 1, verifica 1 bit a partir da posição do mesmo, pula uma casa, verifica mais 1 bit e assim sucessivamente até o final do pacote, a 2, verifica 2 bits a partir da posição do mesmo, pula duas casas, verifica mais 2 bits, etc). Para designar o valor, é feita a soma dos bits, se soma for par, então bit de paridade é 0, se for ímpar, é 1.

Na verificação se compara os mesmos bits relacionados a suas paridades, se bit de paridade for 0 então o resultado tem que ser par, senão, houve um erro que pode ser corrigido caso seja único no pacote, se tiver mais de um erro eles somente são detectados.

## 4.2 Implementação e execução

Este projeto foi desenvolvido em Python 3.7.0. Portanto, faz-se necessário que esta versão ou uma mais recente esteja instalada.

## 4.3 Executando

```
1 $ py hamming.py <qtd_bits_dados> <reps> <prob. erro>
```

Onde os parâmetros são, respectivamente: quantidade de bits de dados do pacote, número de repetições e probabilidade de erro.

Veremos, a seguir, uma breve descrição das principais funções utilizadas

#### *4.3.0.1 Função geomRand(p)*

Gera um numero pseudo-aleatorio com distribuicao geometrica.

#### *4.3.0.2 Função insertErrors(codedPacket, errorProb)*

Insere erros no pacote codedPacket com probabilidade errorProb.

#### *4.3.0.3 Função generateRandomPacket(l)*

Gera um pacote aleatório, passando a quantidade de bits de dados.

#### *4.3.0.4 Função countErrors(originalPacket, decodedPacket)*

Conta a quantidade de erros inseridos, comparando os bits do pacote original com os bits do pacote decodificado.

#### *4.3.0.5 Função numeroBitsParidade(originalPacket)*

Determina a quantidade de bits de paridade, recebendo os bits de dados como parâmetro.

#### *4.3.0.6 Função insereEspacosParaBitsParidade(originalPacket)*

Cria um novo pacote, com espaços para os bits de paridade e com os bits de dados em suas devidas posições.

#### *4.3.0.7 Função hamming(dados)*

Cria um novo pacote, já com os valores finais dos bits de paridade.

#### *4.3.0.8 Função hammingCorrecao(codedPacketComErros)*

Função para decodificação e correção do pacote recebido.

# 5 ANÁLISE DOS CÓDIGOS

1. **Como os diferentes parâmetros afetam a eficiência dos métodos implementados?**

   Quanto maior o tamanho do pacote, mais bits errados podem ser inseridos, quanto maior o número de repetições menor proporcionalmente é a taxa de erros e quanto maior a probabilidade de erro, maior a taxa de erros de bits.

| Tamanho do pacote | 10 | 10 | 100 | 100 |
|---|---|---|---|---|
| Repetições | 10000 | 1000 | 1000 | 10000 |
| Probabilidade de erro | 0,50% | 1% | 0,50% | 1% |
| Erros inseridos | 394 | 813 | 4026 | 7814 |
| Taxa de erro de bits (antes da decodificação) | 49,25% | 1,02% | 50,33% | 97,68% |
| Número de bits corrompidos após decodificação | 394 | 813 | 4026 | 7814 |
| Taxa de erro de bits (após decodificação) | 49,25% | 1,02% | 50,33% | 97,68% |
| Número de pacotes corrompidos | 320 | 558 | 985 | 1000 |
| Taxa de erro de pacotes | 32,00% | 55,80% | 98,50% | 100,00% |

**Tabela 1** - Sem FEC

| Tamanho do pacote | 4 | 4 | 6 | 6 | 9 | 9 |
|---|---|---|---|---|---|---|
| Repetições | 10000 | 1000 | 1000 | 10000 | 10000 | 1000 |
| Probabilidade de erro | 1% | 0,50% | 1% | 0,50% | 1% | 0,50% |
| Erros inseridos | 3160 | 150 | 680 | 3325 | 13550 | 667 |
| Taxa de erro de bits (antes da decodificação) | 98,75% | 46,88% | 103,03% | 50,38% | 1,00% | 49,41% |
| Número de bits corrompidos após decodificação | 0 | 0 | 1 | 1 | 2 | 0 |
| Taxa de erro de bits (após decodificação) | 0,00% | 0,00% | 0,21% | 0,02% | 0,03% | 0,00% |
| Número de pacotes corrompidos | 0 | 0 | 1 | 1 | 1 | 0 |
| Taxa de erro de pacotes | 0,00% | 0,00% | 0,10% | 0,01% | 0,01% | 0,00% |

**Tabela 2** - Paridade bidimensional

| Tamanho do pacote | 1 | 4 | 4 | 11 | 26 | 57 |
|---|---|---|---|---|---|---|
| Repetições | 10000 | 1000 | 1000 | 10000 | 10000 | 1000 |
| Probabilidade de erro | 0,50% | 0,50% | 1,00% | 0,50% | 0,50% | 0,50% |
| Erros inseridos | 315 | 34 | 68 | 83 | 165 | 282 |
| Taxa de erro de bits (antes da decodificação) | 50,00% | 48,57% | 97,14% | 55,33% | 53,23% | 44,76% |
| Número de bits corrompidos após decodificação | 1 | 1 | 4 | 8 | 18 | 25 |
| Taxa de erro de bits (após decodificação) | 1,25% | 0,31% | 1,25% | 0,91% | 0,87% | 0,55% |
| Número de pacotes corrompidos | 1 | 1 | 1 | 1 | 1 | 1 |
| Taxa de erro de pacotes | 0,10% | 0,10% | 0,10% | 0,10% | 0,10% | 0,10% |

**Tabela 3** - Código de Hamming

2. **O quão eficazes esses métodos são em relação a uma solução sem utilização de mecanismos de codificação?**

Por possuírem bits de controle/redundância, os métodos de correção de erros conseguem ser mais eficientes do que os métodos que não o implementam, pois são capazes de detectar e corrigir erros de bits, sem que seja necessário retransmitir o pacote (em determinados casos).

| **Dados** | **noFEC** | **Bidimensional** | **Hamming** |
|---|---|---|---|
| Tamanho do pacote | 10 | 9 | 11 |
| Repetições | 1000 | 1000 | 1000 |
| Probabilidade de erro | 0,50% | 0,50% | 0,50% |
| Erros inseridos | 394 | 667 | 83 |
| Taxa de erro de bits (antes da decodificação) | 49,25% | 49,41% | 55,33% |
| Número de bits corrompidos após decodificação | 394 | 0 | 8 |
| Taxa de erro de bits (após decodificação) | 49,25% | 0,00% | 0,91% |
| Número de pacotes corrompidos | 320 | 0 | 1 |
| Taxa de erro de pacotes | 32,00% | 0,00% | 0,10% |

**Tabela 4** - Comparação entre métodos

3. **Qual método (paridade bidimensional ou Codificação de Hamming) foi mais eficiente em geral?**

   Como pode ser verificado nas tabelas 1, 2 e 3, a paridade bidimensional foi mais eficiente, tendo uma taxa de erro de pacotes inferior ou igual a 0,1%. O código de Hamming, por sua vez, não conseguiu tacas menores que 0,1%.

4. **É possível estabelecer uma relação entre o percentual de overhead introduzido (i.e., o quanto o pacote cresce com a adição de paridade) e a taxa de erro de bits resultante para cada um dos dois métodos?**

   Na paridade bidimensional, utilizou-se matrizes com as seguintes dimensões: 2x2, 2x3 e 3x3, conforme tabela 2. Portanto, em cada uma delas, foram adicionados 4, 5 e 6 bits de paridade, respectivamente, o que representa um aumento de 50%, 36% e 26% nos tamanhos dos pacotes codificados.

   No código de Hamming, foram transmitidos pacotes com 1, 4, 11, 26 e 57 bits de dados, como mostra a tabela 3. Para cada um destes pacotes, foram adicionados, respectivamente, 2 bits de paridade (66,6% da carga total), 3 (42,8%), 4 (26,6%), 5 (16,1%), 6 (9,5%).

   Mesmo adicionando um maior overhead, a paridade Bidimensional ainda mostrou-se mais eficaz.

# 6 BIBLIOGRAFIA


- Código de Hamming - Codificação, Decodificação e Correção - https://www.youtube.com/watch?v=jmcWNPbsrD4
- Python function for generating hamming code and detecting single bit error for any size of data length - https://gist.github.com/vatsal-sodha/f8f16b1999a0b5228143e637d617c797
- Módulo Códigos: código de Hamming - http://eaulas.usp.br/portal/video.action?idItem=7727
- Código de Hamming - https://pt.wikipedia.org/wiki/C%C3%B3digo_de_Hamming