|  |  |
| --- | --- |
| LogoISCTE-IUL-1 | **Departamento de Ciências e Tecnologias da Informação**  **Redes Digitais I** |

**Trabalho Prático I**

**Detecção e Correcção de Erros**

**Curso Grupo**

**Nº Nome \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Nº Nome \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Nº Nome \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Nº Nome \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Ano Lectivo 2012/2013**

**Declaração**

Os alunos abaixo-assinado declaram que o trabalho a que se refere o relatório que a seguir se apresenta é original e foi realizado exclusivamente por si.

**Nº Ass. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Nº Ass. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Nº Ass. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Nº Ass. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Data**

Índice

[Introdução 4](#_Toc354134797)

* [Bit de paridade 4](#_Toc354134798)
* [CRC (Controlo de Redundância Cíclico) 4](#_Toc354134799)
* [Códigos de Hamming 4](#_Toc354134800)
* [**Expressões teóricas** 5](#_Toc354134801)
* [**Expressões experimentais** 6](#_Toc354134802)

[Implementação 7](#_Toc354134803)

[*Funcionamento teórico das técnicas implementadas* 7](#_Toc354134804)

* [Bit de paridade 7](#_Toc354134805)
* [Controlo de Redundância Cíclico (CRC) 7](#_Toc354134806)
* [Códigos de Hamming 8](#_Toc354134807)

[*Funcionamento prático das técnicas implementadas* 10](#_Toc354134808)

[Resultados da Simulação 11](#_Toc354134809)

[Análise dos Resultados Obtidos 15](#_Toc354134810)

[Conclusões 16](#_Toc354134811)

[Anexos 17](#_Toc354134812)

[Referências 17](#_Toc354134813)

[Código fonte 18](#_Toc354134814)

# Introdução

Este trabalho consiste na detecção e correcção de erros que possam existir numa transferência de informação nas ligações de dados. A existência de erros nas comunicações deve-se ao facto de haver interferências nos canais de transmissão, como no caso deste trabalho prático, trata-se de um canal ruidoso, que por si irá provocar erros (segundo a probabilidade de erro de bit) na transmissão das tramas enviadas.

O objectivo principal deste trabalho prático consiste na implementação e teste de técnicas de detecção e correcção de erros. Iremos explicar sumariamente o funcionamento de cada técnica:

## Bit de paridade

A técnica de detecção de erros por bit de paridade consiste basicamente em adicionar um bit à mensagem enviada. Como pedido, usámos a paridade par, ou seja, quando o número de 1’s é par, adiciona um ‘0’ no início da mensagem. Caso seja ímpar, adiciona um ‘1’ no inicio da mensagem. Esta técnica detecta apenas os erros em número ímpar, e não permite a correcção dos erros encontrados.

## CRC (Controlo de Redundância Cíclico)

Esta técnica consiste apenas na detecção de erros, em que o emissor e o receptor concordam com um polinómio gerador G(x), em que quanto maior for o grau do polinómio, maior será a eficácia na detecção de erros. O polinómio é conhecido pelo emissor e pelo seu receptor. É dado pela seguinte expressão: G(x) = x4 + x2 + x + 1, em que r é o grau do polinómio.

## Códigos de Hamming

Esta técnica, ao contrário das outras duas técnicas supra-apresentadas, permite também a correcção de erros. Sabe-se que para qualquer código de Hamming, a sua distância de Hamming é 3 (d=3), logo esta técnica:

* Detecta de certeza até 2 erros.
* Corrige correctamente apenas 1 erro.

Esta técnica consiste na criação de códigos que detectam e corrigem erros, mantendo a mensagem original intacta (caso seja corrigido correctamente). Usámos para o efeito, os códigos Hamming(N,n), em que N representa o código de Hamming que é formado a partir da mensagem original com n bits.

Para fins de tratamento de resultados, que consiste na comparação dos resultados teóricos com os resultados experimentais, usámos as expressões abaixo apresentadas:

### Expressões teóricas:

**- Probabilidade de transmissão sem erros: (1.1)**

**- Valor esperado de número de bits errados: (1.2)**

**- Probabilidade de não detecção de erros, sabendo que houve erros: (1.3)**

**Bit paridade: (1.3.1)**

**-** Valor exacto: **(1.3.1.a)**

**-** Valor aproximado: **(1.3.1.b)**

**Controlo de Redundância Cíclico (CRC): (1.3.2)**

**-** Valor exacto: **(1.3.2.a)**

**-** Valor aproximado: **(1.3.2.b)**

**Códigos de Hamming: (1.3.3)**

**-** Valor exacto: **(1.3.3.a)**

**-** Valor aproximado: **(1.3.3.b)**

**- Probabilidade de correcção correcta, sabendo que houve erros: (1.4)**

**Códigos de Hamming:**

### Expressões experimentais

**- Percentagem de transmissão sem erros: (2.1)**

**- Valor de bits errados: (2.2)**

**- Percentagem de erros não detectados, sabendo que houve: (2.3)**

**- Percentagem de erros corrigidos correctamente (Códigos de Hamming): (2.4)**

**Legenda:**

**- *p:*** Probabilidade de erro de bit independente;

**- *N:*** Número de bits na trama;

**- *T:*** Número de tramas transmitidas;

**- *PSE:*** Probabilidade de transmissão sem erros;

**- *P1E:*** Probabilidade de existir 1 erro na trama;

**- *P2E:*** Probabilidade de existir 2 erros na trama;

**- *Ai*:** Factor multiplicador de k (Ver tabelas 1.1 e 1.2 em anexo);

**- i:** Iterador do somatório;

**- r:** Grau do polinómio gerador;

**- *Pares*:** Conjunto de todos os números pares positivos (>0).

# Implementação

### Funcionamento teórico das técnicas implementadas

## Bit de paridade

Como já foi dito na ‘Introdução’, a técnica do bit de paridade (paridade par) consiste na detecção de erros em número ímpar numa trama. O emissor adiciona um bit à mensagem, que no nosso caso encontra-se no início da mensagem. Se o número de 1’s da mensagem for ímpar, então adiciona-se o bit‘1’. Caso contrário, o bit ‘0’. Esta técnica permite apenas a detecção de 50% dos erros possíveis de encontrar numa trama.

## Controlo de Redundância Cíclico (CRC)

Sabendo que G(x) = x4 + x2 + x + 1, em que r é o grau do polinómio, temos que saber que:

* + O polinómio P(x) é formado pela palavra original concatenada com os bits de paridade (neste caso são 4 zeros, porque r = 4), e deve ser também divisível por G(x).
  + O emissor, por si, tenta dividir P(x) por G(x) (usando operações XOR).
  + Finda a divisão, é formada a trama T(x) que é simplesmente a mensagem original concatenada com o FCS (Frame Check Sequence, que neste caso é o resto da divisão de P(x) por G(x) e é constituído por r bits).
  + Depois de formada a trama T(x), esta vai ser transmitida através do canal ruidoso, o que irá (ou não) formar erros na trama. Para verificar a existência desses erros, o receptor divide T(x) pelo polinómio G(x), e caso o resto da divisão R(x) for zero, então não houve erros.
  + Por consequência, cada ‘1’ no R(x) corresponde a um bit invertido.

A figura abaixo representa o diagrama do registo de deslocamento que implementa o polinómio gerador proposto.

C0

C2

C3

C1

Input

*Fig. 1: Diagrama de deslocamento do registo para o código CRC.*

Todos os registos começam a ‘0’ aquando a emissão. A mensagem entra bit a bit, começando com o bit mais significativo. Efectua-se as operações XOR respeitando o diagrama. Depois do último bit ter sido processado o registo de deslocamento contém a FCS. Na recepção utiliza-se o mesmo circuito. À medida que os bits vão chegando, neste caso os bits da trama recebida, são inseridos no registo de deslocamento. No final do procedimento, se não tiverem ocorrido erros, ou não tiverem sido detectados, todo o registo deve estar a ‘0’.

## Códigos de Hamming

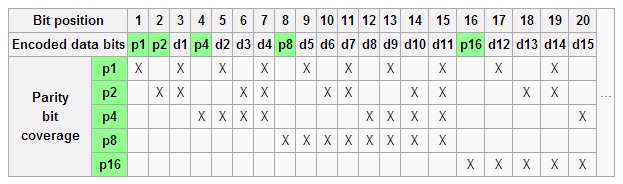
Para podermos codificar a nossa mensagem, temos de ter em conta o número de bits existentes na trama. No nosso caso, usámos os códigos Hamming(7,4) e Hamming(15,11).

Dada uma mensagem D, e os bits de paridade P, temos:

* **Para n = 4 bits**

O código de Hamming para n = 4 bits irá ter 7 bits no total, ou seja é composto por 4 bits de dados e 3 bits de paridade.

Usámos a seguinte tabela como referência para calcular os bits de paridade.



*[2.2] Tabela representativa dos bits de paridade para os códigos de Hamming.*

Para podermos ter os valores dos bits de paridade, teremos de os calcular recorrendo a operações XOR:

* **Para n = 11 bits**

O código de Hamming para n = 11 bits irá ter 15 bits no total, ou seja é composto por 11 bits de dados e 4 bits de paridade.

Usando a tabela [2.2], calculamos os valores dos bits de paridade recorrendo a operações XOR:

* **Correcção de erros usando ‘check bits’**

Para proceder à correcção de erros temos de determinar os ‘check bits’ que nos vão dizer a posição do erro, para que se proceda à correcção (inversão do bit errado).

* **Para n = 4 bits**
* **Para n = 11 bits**

Temos de ter em atenção que o código de Hamming só corrige correctamente um erro.

### Funcionamento prático das técnicas implementadas

Para a implementação dos 3 métodos de detecção / correcção de erros, foram criadas duas classes com o propósito de servir de bibliotecas de utilidade para o trabalho, nomeadamente *CRCUtil* e *HammingUtil* para os métodos de CRC e Código de Hamming, respectivamente, e a classe de Motor que se serve de métodos para implementar o Bit de Paridade.

* **HammingUtil.java**

Dentro da classe *HammingUtil* encontram-se os métodos que implementam o código de Hamming, na forma generalizada para (7,4) e (15,11). Estes tomam proveito das tabelas para chegar aos bits de paridade e check bits com XORs.

É de realçar o método *String hamming(String codeStr)* que produz o código de hamming de uma *String* de dados (em bits) e retorna-a e o método *int countCheck(String codeStr)* que retorna a posição em que existe um erro detectável.

Esta classe inclui também o método *int getDistance(String one, String two)* que retorna o valor da distância de hamming entre duas *Strings* de dados em bits. Este é usado para calcular o número de bits errados por trama enviada no simulador.

* **CRCUtil.java**

A classe *CRCUtil* é uma biblioteca que inclui os métodos de geração dos códigos CRC a partir dos dados e o método de detecção de erro.

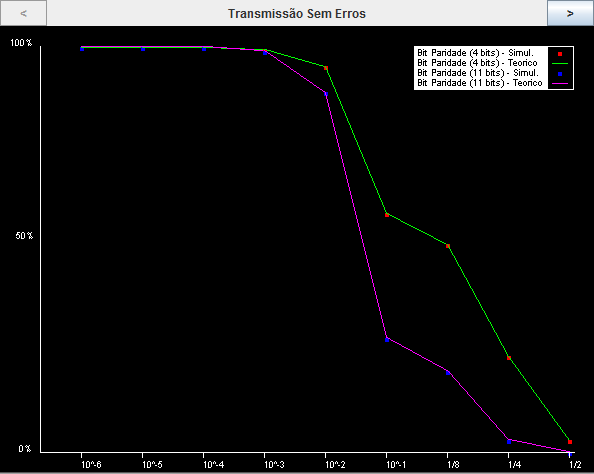
*String crc(String message, String divisorOrig)* é o método gerador CRC, tem como input o *String* de dados em bits e *String* com o polinómio gerador em bits. *Boolean* *detectErrors(String remStr)* é o método que retorna *true* caso a mensagem contenha um erro, e falso caso não detecte.

* **MotorDemonstracao.java**

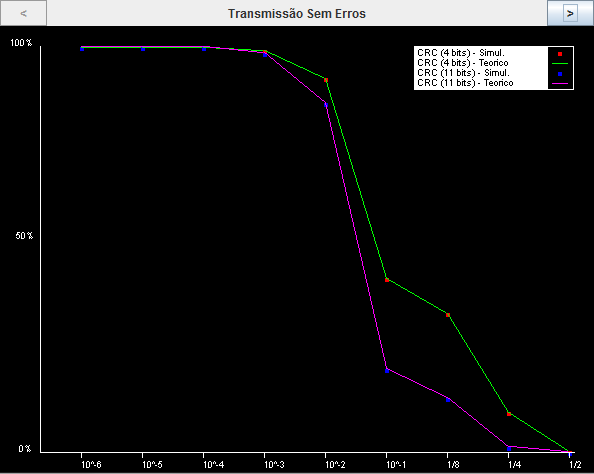
MotorDemonstracao é a classe que inclui os métodos de Bit de Paridade, faz os cálculos necessários para o modo de demonstração como a geração de erros com probabilidade por bit e inserção do padrão de erros numa trama.

O método que calcula o bit de paridade é *String paridade(String in)* que toma como input a palavra em bits e retorna um único bit (de paridade). Para a verificação usa-se *boolean detectErrorsParidade(String after)* que retorna *true* caso haja erro, falso caso não detecte.

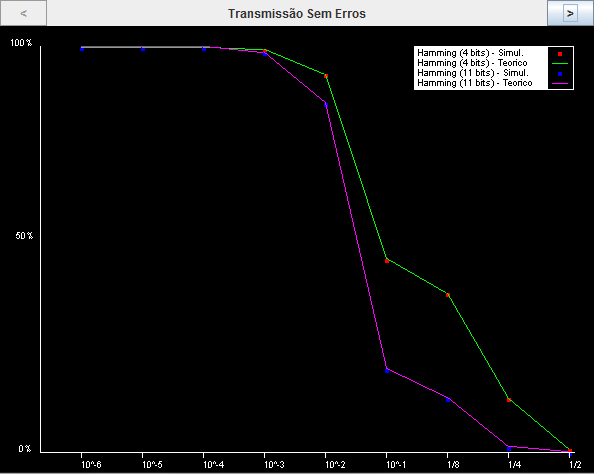
# Resultados da Simulação



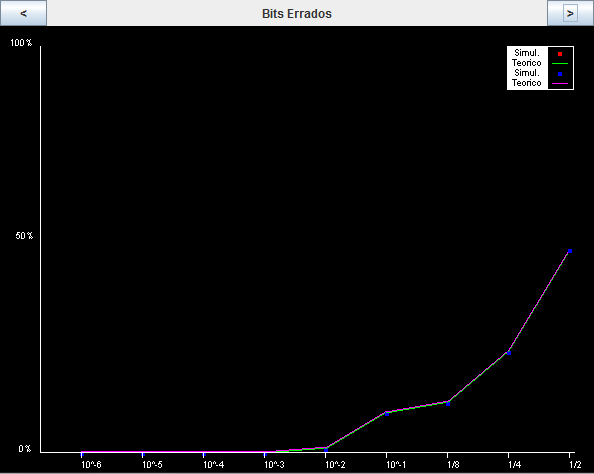
*Gráfico 1. Percentagem Experimental e Probabilidade Teórica de Transmissão sem Erros com o método de Bit de Paridade com 4 e 11 bits.*



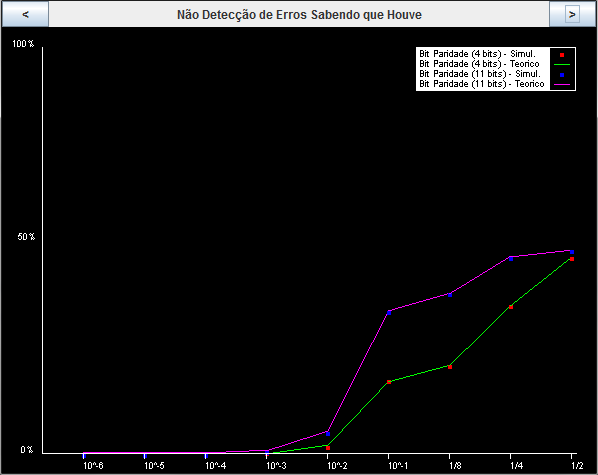
*Gráfico 2. Percentagem Experimental e Probabilidade Teórica de Transmissão sem Erros com o método de CRC com 4 e 11 bits.*

**

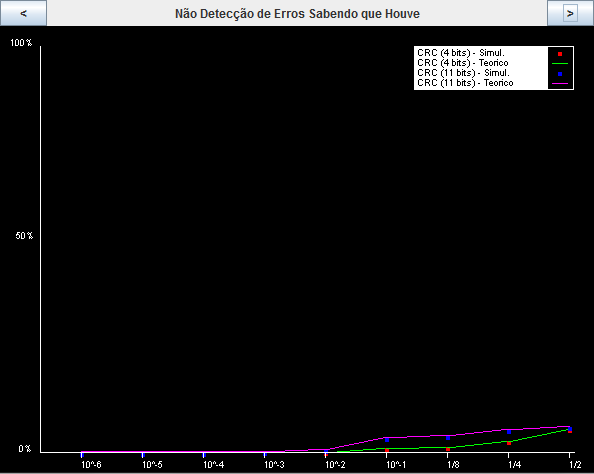
*Gráfico 3. Percentagem Experimental e Probabilidade Teórica de Transmissão sem Erros com o método de Código de Hamming com 4 e 11 bits.*

**

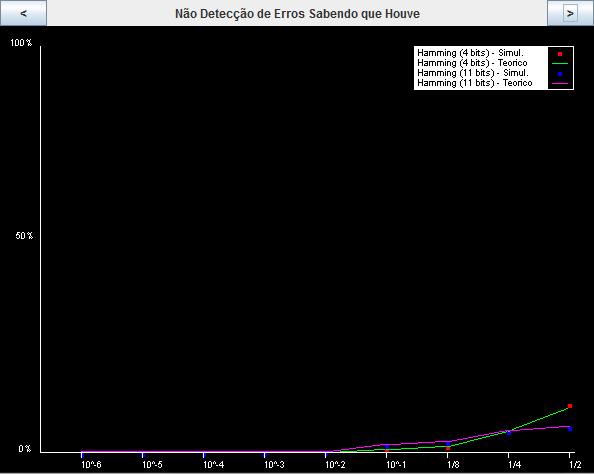
*Gráfico 4. Percentagem Experimental e Probabilidade Teórica do número de Bits Errados por Trama com 4 e 11 bits.*

**

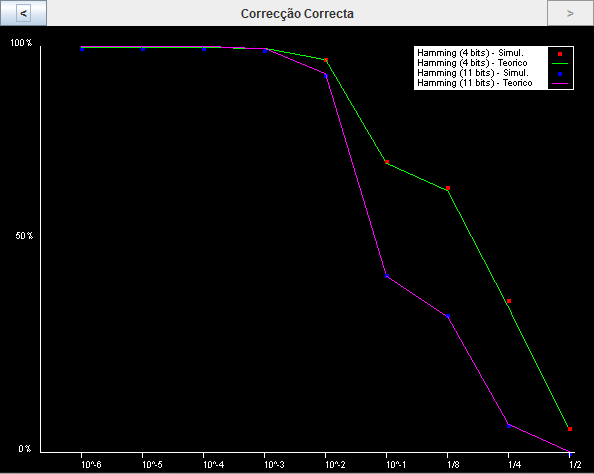
*Gráfico 5. Percentagem Experimental e Probabilidade Teórica de Não Detecção de Erros sabendo que houve com o método de Bit de Paridade com 4 e 11 bits.*

**

*Gráfico 6. Percentagem Experimental e Probabilidade Teórica de Não Detecção de Erros sabendo que houve com o método de CRC com 4 e 11 bits.*

**

*Gráfico 7. Percentagem Experimental e Probabilidade Teórica de Não Detecção de Erros sabendo que houve com o método de Código de Hamming com 4 e 11 bits.*

**

*Gráfico 8. Percentagem Experimental e Probabilidade Teórica de Correcção Correcta de uma Trama detectada incorrecta (Código de Hamming) com 4 e 11 bits.*

# Análise dos Resultados Obtidos

Os gráficos 1, 2 e 3 representam as probabilidades de transmissão sem erros em função da probabilidade de erro por bit independente para as técnicas de Bit Paridade, CRC e Códigos Hamming, respectivamente. Podemos concluir que os resultados para cada uma das técnicas são semelhantes, apesar de se notarem pequenas diferenças. Ambas as técnicas CRC e Códigos de Hamming apresentam a mesma percentagem de transmissão sem erros para 11 bits, mas para 4 bits, a técnica de Códigos de Hamming é consideravelmente melhor. Quanto ao Bit Paridade, tem uma melhor probabilidade de transmissão sem erros do que as outras duas técnicas em ambos 4 e 11 bits. Essas diferenças provêm do facto das tramas na sua totalidade serem menores em número de bits na técnica Bit de Paridade (5 e 12 bits por trama) do que nas outras técnicas (CRC – 8 e 15 bits por trama, Hamming – 7 e 15 bits por trama). E como se pode verificar na fórmula 1.1, a probabilidade de transmissão sem erros apenas depende da probabilidade de erro por bit e do número de bits por trama aquando a transmissão.

O gráfico 4 representa a percentagem de bits errados totais na transmissão de **T** tramas. Este valor é igual para todos os casos, pois representa apenas uma percentagem que é exactamente igual à probabilidade de erro de bit. Isto deve-se ao facto do valor esperado de números de bits errados (fórmula 1.3) ser igual à multiplicação entre o número de bits por trama **N**, o total de tramas transmitidas **T** e a probabilidade de erro de bit **P**, tudo a dividir pelo número total de bits transmitidos, que é simplesmente **N** x **T**, ficando apenas **P**.

Os gráficos 5, 6 e 7 representam a probabilidade teórica e a percentagem experimental de não detecção de erros, sabendo que houve (probabilidades condicionadas).

Para a técnica Bit de Paridade notamos uma subida a partir de **P = 10-2**, sendo que a probabilidade é consideravelmente maior para tramas de 11 bits, tendendo para 50% à medida que a probabilidade de erro de bit tende para **P = 1/2**. Para as duas técnicas CRC e Hamming, temos uma probabilidade de não detecção de erros muito baixas, sendo que o Código de Hamming tem uma percentagem menor até **P = 1/8** e a partir da mesma, tem uma subida em tramas com 4 bits, com **P = 1/2**, até aproximadamente 11%. Na técnica CRC, para 11 bits por trama, a probabilidade é ligeiramente maior (1%) do que probabilidade obtida na técnica de Código de Hamming, mas sendo que a partir de **P = 1/8**, a subida da probabilidade é linear (ao contrário da técnica Hamming).

O gráfico 8 representa a probabilidade teórica e a percentagem experimental de correcção correcta, sabendo que houve erros na trama (probabilidade condicionada). Para todos os **P ≤ 10-2**, a probabilidade de correcção correcta mantém-se sempre acima dos 95% porque como o código de Hamming corrige apenas correctamente 1 erro, a probabilidade de haver mais do que 1 erro é muito pequena. À medida que a probabilidade de erro por bit aumenta, nota-se uma descida substancial na percentagem de correcções correctas (tendendo para 0% em **P = 1/2**), pois existe muito maior probabilidade de haver mais do que 1 erro por trama.

# Conclusões

Em geral, tomando partido de todas as técnicas utilizadas neste projecto de simulação e analisando todos os dados obtidos com cuidado, conseguimos estabelecer a diferença entre cada técnica, podendo estipular os casos em que se deve utilizar cada uma.

Para experiências reais, a melhor técnica a ser utilizada para nas transmissões é o Código de Hamming. Este, para além de ser melhor a detectar erros, também conseguir corrigir correctamente alguns deles. Pois na realidade, as probabilidades de erro de bit não deverão ser tão elevadas como foram neste projecto, nomeadamente todas acima de **P = 10-2** são irrealistas.

Considerando os resultados da simulação, a técnica Bit de Paridade é nos apresentada como sendo a melhor para transmissões sem erros, pois como já foi visto anteriormente, essa probabilidade depende do número de bits por trama, e esta técnica apenas adiciona 1 bit de paridade a cada trama. Por outro lado, a técnica de CRC acrescenta sempre 4 bits de paridade, fazendo com que seja a menos eficiente neste caso.

Como nota pessoal, foi-nos gratificante a realização deste projecto, pois permitiu fundamentar bem os conhecimentos adquiridos nesta cadeira.

# Anexos

[1.1] Tabela dos padrões de erros não detectados para mensagens de 4 e 11 bits.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Número de padrões de erros não detectados correspondendo a *i* bits errados para o polinómio proposto (CRC) | | | | | | | | | | | | | | |  |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |  |
|  | ***m = 4*** | 0 | 1 | 0 | 11 | 0 | 3 | 0 | 0 | - | - | - | - | - | - | - |  |
|  | ***m = 11*** | 0 | 9 | 0 | 177 | 0 | 625 | 0 | 795 | 0 | 387 | 0 | 51 | 0 | 3 | 0 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Número de padrões de erros não detectados correspondendo a *i* bits errados para o polinómio proposto (Códigos de Hamming) | | | | | | | | | | | | | | |  |
|  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |  |
|  | ***m = 4*** | 0 | 0 | 7 | 7 | 0 | 0 | 7 | - | - | - | - | - | - | - | 0 |  |
|  | ***m = 11*** | 0 | 0 | 35 | 105 | 168 | 280 | 435 | 435 | 280 | 168 | 105 | 35 | 0 | 0 | 1 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

# Referências

[2.1] Wickerhauser, Mladen V.; “Mathematics for Multimedia”; Applied and Numerical Harmonic Analysis; eBook: “http://books.google.pt/books?id=ed9KSMW7uL0C”.

[2.2] Tabela representativa dos bits de paridade para os códigos de Hamming.

Link: http://math.stackexchange.com/questions/355934/hamming-code-error-correction

[2.3] Fórmula (1.3.1.a): Probabilidade de não detecção de erros, sabendo que houve.

Link:http://math.stackexchange.com/questions/79484/expressing-probability-that-successes-is-an-even-number-mathematically

# Código fonte

* MotorSim.java

public class MotorSim extends JComponent implements Runnable {

private static final long *serialVersionUID* = 1L;

**/\***

**Tabelas para calcular as probabilidades de não detecção de Erro com os métodos de CRC e Código de Hamming.**

**\*/**

private static final int[] *crcErrorFour* = {1,11,3}, *crcErrorEleven* = {9,177,625,795,387,51,3};

private static final int[] *hammingErrorFour* = {0,0,7,7,0,0,7}, *hammingErrorEleven* = {0,0,35,105,168,280,435,435,280,168,105,35,0,0,1};

**// Tabela de probabilidades (escolha para gráficos)**

private static final double[] *probs* = {0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1, 0.125, 0.25, 0.5};

private JFrame janela;

private int numBits, numTramas, errorMethod, tramasTransmitted, tramasComErro, tramasArranjadas, tramasArranjadasMal,tramasComErroNaoDetectados, bitsErradosExp;

private int selProb;

private double selectedProb;

private boolean stop = false;

private GraphWindow graph;

private JButton stopButton;

public MotorSim(int errorMethod, int numBits, int numTramas, int selProb, JFrame janela, GraphWindow graph, JButton stopButton) {

this.janela = janela;

this.errorMethod = errorMethod;

this.numBits = numBits;

this.numTramas = numTramas;

this.selProb = selProb;

this.selectedProb = *probs*[selProb];

this.graph = graph;

this.stopButton = stopButton;

new Thread(this).start();

}

**/\***

**Para propósitos de facilidade de manobra no código, os cálculos teóricos são realizados à medida que nos são apresentados**

**os valores experimentais da simulação. Daí estarem dentro do método paintComponent.**

**Cada vez que é redesenhado (a cada aprox. 100 mil tramas enviadas), apresenta os valores teoricos probabilisticos**

**e as percentuagens experimentais.**

**\*/**

protected void paintComponent(Graphics g) {

g.setColor(Color.*BLACK*);

Rectangle size = g.getClipBounds();

g.fillRect(0,0,size.width,size.height);

g.setColor(Color.*GREEN*);

g.setFont(new Font("Arial",Font.*BOLD*,12));

g.drawString("SIMULAÇÃO", 10, 19);

String metd = "BIT PARIDADE";

if(errorMethod == 1)

metd = "CRC";

if(errorMethod == 2)

metd = "HAMMING";

g.drawString("ERROR DETECTION METHOD: "+metd, 10, 32);

g.drawString("#BITS: "+numBits, 10, 45);

g.drawString("ERROR PROBABILITY p/ BIT: "+selectedProb, 10, 58);

g.setColor(Color.*RED*);

g.drawString("----------------------------", 10, 71);

g.setColor(Color.*GREEN*);

double ntmpBits = (double) numBits;

if(errorMethod == 0){

ntmpBits = numBits + 1;

} else if(errorMethod == 1){

ntmpBits = numBits + 4;

} else if(errorMethod == 2){

if(numBits == 4){

ntmpBits = 7;

} else {

ntmpBits = 15;

}

}

g.drawString("#BITS p/ TRAMA: "+(int)ntmpBits, 420, 19);

g.setColor(Color.*YELLOW*);

g.drawString("THEORICAL STATS", 10, 84);

g.setColor(Color.*WHITE*);

**/\***

**As probabilidades foram calculadas a partir das fórmulas estipuladas no relatório.**

**Probabilidade de Transmissão sem Erros Teórica.**

**Em que Ptse = (1-Pb)^L, sendo Pb (selectedProb a probabilidade de erro por bit escolhida**

**e ntmpBits (L) o numero de bits na trama.**

**\*/**

double ptseTeorico = Math.*pow*((1-selectedProb),ntmpBits);

**/\***

**Numero esperado de bitsErrados no total (numero de bits por trama \* numero de tramas no total \* probabilidade de erro por bit)**

**\*/**

int bitsErrados = (int) (ntmpBits \* numTramas \* selectedProb);

double pndceTeorico = 0;

double pdccTeorico = 0;

**/\***

**Para o cálculo da Probabilidade de Não Detecção, sabendo que houve erro,**

**recorremos a vários métodos que se provam eficazes na previsão da probabilidade, para cada**

**uma das técnicas utilizadas.**

**Para o bit de paridade toma-se partido a fórmula simplificada, explicada no relatório.**

**No caso das técnicas de CRC e Código de Hamming,**

**o método utilizado baseia-se no uso das tabelas (listadas no topo da classe)**

**que permitem, graças à pre-análise extensiva, chegar aos valores exactos para o cálculo.**

**Foi utilizado o valor exacto para os dois, apesar de ter sido recomendado a aproximação no caso do código de hamming.**

**\*/**

if(errorMethod == 0){ //Bit paridade

pndceTeorico = (0.5)\*(1+(Math.*pow*(1-2\*selectedProb, ntmpBits)) - 2\*Math.*pow*(1-selectedProb, ntmpBits));

pndceTeorico /= (1.0 - ptseTeorico);

} else if(errorMethod == 1){ //CRC

int k = 2;

int[] tempCrcError;

if(numBits == 4){

tempCrcError = *crcErrorFour*;

} else {

tempCrcError = *crcErrorEleven*;

}

for(int i = 0; i < tempCrcError.length ;i++){

double ptemp = tempCrcError[i] \* Math.*pow*(selectedProb, k) \* Math.*pow*(1.0-selectedProb, ntmpBits - k);

pndceTeorico += ptemp;

k+=2;

}

pndceTeorico /= (1.0 - ptseTeorico);

} else if(errorMethod == 2){ //Codigo Hamming

//pndce

int[] tempHammingError;

if(numBits == 4){

tempHammingError = *hammingErrorFour*;

} else {

tempHammingError = *hammingErrorEleven*;

}

for(int i = 1; i <= tempHammingError.length; i++){

double ptemp = tempHammingError[i-1] \* Math.*pow*(selectedProb, i) \* Math.*pow*(1.0-selectedProb, ntmpBits-i);

pndceTeorico += ptemp;

}

pndceTeorico /= (1.0 - ptseTeorico);

**/\***

**Probabilidade de correcção correcta, dado que ocorreu um erro. (Probabilidade condicionada pela probabilidade de haver erro)**

**\*/**

pdccTeorico = ntmpBits \* selectedProb \* Math.*pow*(1.0-selectedProb,ntmpBits-1) / (1.0 - ptseTeorico);

}

g.drawString("Prob. Transmissao sem Erros: "+ptseTeorico\*100+"%", 10, 97);

g.drawString("Valor esperado de #bits errados: "+bitsErrados, 10, 110);

g.drawString("Valor esperado de Percentagem de Bits Errados: "+((double)bitsErrados/(double)(numTramas\*ntmpBits))\*100+"%", 10, 123);

g.drawString("Prob. N Deteçao de Erros Sabendo que Houve: "+pndceTeorico\*100+"%", 10, 136);

if(errorMethod == 2){

g.drawString("Prob de Correçao Correcta: "+pdccTeorico\*100+"%", 10, 149);

}

g.setColor(Color.*CYAN*);

g.drawString("#TRAMAS: "+tramasTransmitted, 10, 175);

g.drawString("#TRAMAS w/ ERRORS: "+tramasComErro, 10, 188);

g.drawString("#TRAMAS w/ ERRORS (NOT DETECTED): "+tramasComErroNaoDetectados, 10, 201);

if(errorMethod == 2){

g.setColor(Color.*GREEN*);

g.drawString("#TRAMAS SUCCESSFULLY FIXED: "+tramasArranjadas, 10, 214);

g.setColor(Color.*RED*);

g.drawString("#TRAMAS ERRANOUSLY FIXED: "+tramasArranjadasMal, 10, 227);

}

g.setColor(Color.*YELLOW*);

g.drawString("EXPERIMENTAL STATS", 10, 253);

g.setColor(Color.*WHITE*);

**//Nota: TramasComErro + TramasComErroNaoDetectados = Todos os Erros**

g.drawString("Percentagem de Transmissão s/ Erros: "+(1.0-((double)(tramasComErro+tramasComErroNaoDetectados)/(double)tramasTransmitted))\*100+"%", 10, 266);

g.drawString("Bits Errados: "+bitsErradosExp, 10, 279);

g.drawString("Percentagem de Bits Errados: "+((double)bitsErradosExp/(double)(tramasTransmitted\*ntmpBits))\*100+"%", 10, 292);

g.drawString("Percentagem de Erros Não Detectados: "+((double)tramasComErroNaoDetectados/(double)(tramasComErro+tramasComErroNaoDetectados))\*100+"%", 10, 305);

if(errorMethod == 2){

g.drawString("Percentagem de Tramas Corrigidas Correctamente: "+((double)tramasArranjadas/(double)(tramasComErro))\*100+"%", 10, 318);

}

**//Grafico**

**/\***

**graphIndex:**

**0 - transmissao sem erros**

**1 - bits errados**

**2 - nao detectados**

**3 - corrigidas correctamente**

**\*/**

graph.setGraphVar(0, selProb, ptseTeorico, numBits, errorMethod, false); //transmissao sem erros teorico

graph.setGraphVar(1, selProb, (double)bitsErrados/(double)(numTramas\*ntmpBits), numBits, errorMethod, false); **//bits errados teorico**

graph.setGraphVar(2, selProb, pndceTeorico, numBits, errorMethod, false); //nao detectados teorico

graph.setGraphVar(0, selProb, (1.0-((double)(tramasComErro+tramasComErroNaoDetectados)/(double)tramasTransmitted)), numBits, errorMethod, true); **//transmissao sem erros exp**

graph.setGraphVar(1, selProb, (double)bitsErradosExp/(double)(tramasTransmitted\*ntmpBits), numBits, errorMethod, true); **//bits errados exp**

graph.setGraphVar(2, selProb, (double)tramasComErroNaoDetectados/(double)(tramasComErro+tramasComErroNaoDetectados), numBits, errorMethod, true); **//nao detectados exp**

if(errorMethod == 2){

graph.setGraphVar(3, selProb, pdccTeorico, numBits, errorMethod, false); //corrigidos correctamente teorico

graph.setGraphVar(3, selProb, (double)tramasArranjadas/(double)tramasComErro, numBits, errorMethod, true); **//corrigidos** correctamente exp

}

**//Barra de progresso**

double totalDone = (double)tramasTransmitted/(double)numTramas;

double totalSize = 520;

int curSize = (int) (totalSize \* totalDone);

g.setColor(Color.*YELLOW*);

g.fillRect(10, 420, curSize, 25);

g.setColor(Color.*BLUE*);

g.drawString((int)(totalDone\*100)+"%", (curSize/2), 438);

}

private void endCalc(){

stopButton.setEnabled(false);

janela.repaint();

}

public void stop(){

stop = true;

}

**/\***

**método transmitData()**

**MODO SIMULAÇÃO**

**Utilizado em cada iteração de envio de uma trama, calcula (igual ao modo de demonstração) o processo de passagem de uma trama**

**desde a geração aleatória de Dados em bits, passando pelo processo de inserção de erros baseando-se em probabilidade de bit**

**até a recepção e correcção (quando aplicável) de cada uma.**

**Este processo**

**\*/**

private void transmitData(){

tramasTransmitted++;

String beforeTrans = MotorDemonstracao.*generateRandomData*(numBits);

if(errorMethod == 0){ **//Bit Paridade**

String toTransmit = beforeTrans;

String parity = MotorDemonstracao.*paridade*(toTransmit);

String afterTransmit = MotorDemonstracao.*insertErrorsAuto*(parity+toTransmit, selectedProb);

bitsErradosExp += HammingUtil.*getDistance*(parity+toTransmit, afterTransmit); **// A distancia de hamming da-nos a diferença (em #bits) entre 2 palavras**

if(!MotorDemonstracao.*detectErrorsParidade*(afterTransmit)){

if(!afterTransmit.equals(parity+toTransmit)){

tramasComErroNaoDetectados++;

}

} else {

tramasComErro++;

}

} else if(errorMethod == 1){ **//CRC**

String toTransmit = CRCUtil.*crc*(beforeTrans, "10111");

String afterTransmit = MotorDemonstracao.*insertErrorsAuto*(toTransmit, selectedProb);

bitsErradosExp += HammingUtil.*getDistance*(toTransmit, afterTransmit);

String crcRemainder = CRCUtil.*crcRem*(afterTransmit, "10111");

boolean detectadoErro = CRCUtil.*detectErrorsCRC*(crcRemainder);

if(!detectadoErro){

if(!afterTransmit.equals(toTransmit)){

tramasComErroNaoDetectados++;

}

} else {

tramasComErro++;

}

} else if(errorMethod == 2){ **//Hamming**

String toTransmit = HammingUtil.*hamming*(beforeTrans);

String afterTransmit = MotorDemonstracao.*insertErrorsAuto*(toTransmit, selectedProb);

boolean detectadoErro = HammingUtil.*hasError*(afterTransmit);

bitsErradosExp += HammingUtil.*getDistance*(toTransmit, afterTransmit);

if(!detectadoErro){

if(!afterTransmit.equals(toTransmit)){

tramasComErroNaoDetectados++;

}

} else {

tramasComErro++;

String hammingErrorFix = HammingUtil.*fixError*(afterTransmit);

String hammingOriginal = HammingUtil.*getOriginal*(hammingErrorFix);

if(hammingOriginal.equals(beforeTrans)){

tramasArranjadas++;

} else {

tramasArranjadasMal++;

}

}

}

}

**/\***

**Método run() da implementação Runnable para criação do thread para o simulador.**

**Corre numTramas de vezes ou até que seja comandado parar (flag, boolean stop)**

**\*/**

@Override

public void run() {

tramasTransmitted = 0;

while(tramasTransmitted < numTramas && !stop){

transmitData();

if(tramasTransmitted % (numTramas/100) == 0){

janela.repaint();

}

}

endCalc();

}

}

* MotorDemonstracao.java

public class MotorDemonstracao extends JComponent {

private static final long serialVersionUID = 1L;

private int numBits, errorMethod, hammingErrorPos;

private boolean manual, detectadoErro;

private String bitsDados, padraoErros, toTransmit, afterTransmit, crcRemainder, hammingErrorFix, hammingOriginal;

private double probErro;

**/\***

**Dois métodos constructores, um para a transmissão Manual e outro para a tranmissão Automática**

**A diferença substancial é que no caso manual, inserem-se os dados em bits a enviar e o padrão de erros,**

**possibilitando assim monotorizar o bom funcionamento de cada uma das implementações das técnicas de detecção e correcção de erro.**

**No caso automático, as escolhas limitam-se ao método e ao número de bits de dados a enviar e a probabilidade**

**de erro por bit a inserir na mensagem 'encriptada' (Canal ruidoso). Permitindo assim uma analise mais estatística, apenas**

**com o modo demonstrativo.**

**\*/**

public MotorDemonstracao(int errorMethod, int numBits, String bitsDados, String padraoErros){

this.errorMethod = errorMethod;

this.numBits = numBits;

this.manual = true;

this.bitsDados = bitsDados;

this.padraoErros = padraoErros;

transmitManual();

}

public MotorDemonstracao(int errorMethod, int numBits, double errorProbability){

this.errorMethod = errorMethod;

this.numBits = numBits;

this.manual = false;

this.probErro = errorProbability;

this.bitsDados = generateRandomData(numBits);

transmitAuto();

}

**/\***

**método generateRandomData**

**Gera uma String de dados aleatórios com num bits.**

**\*/**

public static String generateRandomData(int num){

String str = "";

Random rand = new Random();

for(int i = 0; i < num; i++){

if(rand.nextDouble()<=0.5){

str=str+"1";

} else {

str=str+"0";

}

}

return str;

}

**/\***

**método paridade**

**Gera o bit de paridade de dada String de dados em bits.**

**Caso tenha um número par de 1s retorna 0, caso tenha um número ímpar retorna 1.**

**\*/**

public static String paridade(String in){

int ones = 0;

for(int i = 0; i < in.length(); i++){

if(in.charAt(i)=='1')

ones++;

}

if(ones%2==0)

return "0";

return "1";

}

**//método que retorna um XOR entre duas booleanas.**

public static boolean XOR(boolean x, boolean y) {

return ( ( x || y ) && ! ( x && y ) );

}

**/\***

**método insertErrors**

**Serve para inserir erros numa string de dados com base num padrão de erros em bit.**

**Faz XOR entre a mensagem original e o padrão, bit a bit, e retorna o resultado da operação.**

**\*/**

public static String insertErrors(String in, String pattern){

String fin = "";

for(int i = 0; i < pattern.length(); i++){

boolean inBool = false, inPat = false;

if(in.charAt(i) == '1')

inBool = true;

if(pattern.charAt(i) == '1')

inPat = true;

String add = "0";

if(XOR(inBool, inPat))

add = "1";

fin = fin + add;

}

return fin;

}

**//método para inverter um caracter. ('0' fica '1' e '1' fica '0')**

public static char invertBinChar(char in){

if(in == '0')

return '1';

return '0';

}

**/\***

**método insertErrorsAuto**

**Insere erros numa String de dados, baseando-se na probabilidade de erro por bit**

**É usada pelo modo de demonstração e pelo modo de simulação.**

**\*/**

public static String insertErrorsAuto(String in, double prob){

String fin = "";

Random rand = new Random();

for(int i = 0; i < in.length(); i++){

char cur = in.charAt(i);

double r = rand.nextDouble();

if(r <= prob){

cur = invertBinChar(cur);

}

fin=fin+cur;

}

return fin;

}

**/\***

**método detectErrorsParidade**

**Detecta erros com o bit de paridade, retorna true se encontrar um erro**

**retorna false se o bit de paridade se mantiver consistente com a trama.**

**\*/**

public static boolean detectErrorsParidade(String after){

int ones = 0;

char paridade = after.charAt(0);

for(int i = 1; i < after.length(); i++){

if(after.charAt(i)=='1')

ones++;

}

if((ones%2==0 && paridade == '1') || (ones%2!=0 && paridade=='0'))

return true; **//deu erro**

return false; **//nao foram detectados erros.**

}

**/\***

**método transmitManual**

**Simula a tranmissão, baseando-se nas várias técnicas de detecção (e correcção)**

**A String de dados passa pelo cálculo de cada um dos métodos, Bit Paridade, CRC e Hamming para a String toTransmit**

**inserem-se os erros na trama com o método insertErrors (ou seja, manual) directamente com o padrão de erros para a String afterTransmit**

**para todos os casos a booleana detectadoErro vai usar os métodos especificos para cada técnica de transmissão,**

**tentando averiguar com os métodos de detecção se houve um erro.**

**\*/**

private void transmitManual(){

if(errorMethod == 0){ **//Bit paridade**

toTransmit = paridade(bitsDados) + bitsDados;

afterTransmit = insertErrors(toTransmit, padraoErros);

detectadoErro = detectErrorsParidade(afterTransmit);

} else if(errorMethod == 1) { **//CRC**

toTransmit = CRCUtil.crc(bitsDados, "10111"); **//Divisor é sempre 10111**

afterTransmit = insertErrors(toTransmit, padraoErros);

crcRemainder = CRCUtil.crcRem(afterTransmit, "10111");

detectadoErro = CRCUtil.detectErrorsCRC(crcRemainder);

} else if(errorMethod == 2) { **//Hamming**

toTransmit = HammingUtil.hamming(bitsDados);

afterTransmit = insertErrors(toTransmit, padraoErros);

detectadoErro = HammingUtil.hasError(afterTransmit);

hammingErrorPos = HammingUtil.countCheck(afterTransmit);

hammingErrorFix = HammingUtil.fixError(afterTransmit);

hammingOriginal = HammingUtil.getOriginal(hammingErrorFix);

}

}

**/\***

**método transmitAuto**

**Simula a transmissão, parecido ao método transmitManual**

**A String passa pelos mesmos cálculos, mas desta vez os bitsDados vão ser gerados aleatóriamente**

**e o método de inserção de erros utilizado é baseado em probabilidade de erro por bit**

**em vez de um padrão.**

**\*/**

private void transmitAuto(){

if(errorMethod == 0){ **//Bit paridade**

toTransmit = paridade(bitsDados) + bitsDados;

afterTransmit = insertErrorsAuto(toTransmit, probErro);

detectadoErro = detectErrorsParidade(afterTransmit);

} else if(errorMethod == 1) { **//CRC**

toTransmit = CRCUtil.crc(bitsDados, "10111"); **//Divisor é sempre 10111**

afterTransmit = insertErrorsAuto(toTransmit, probErro);

crcRemainder = CRCUtil.crcRem(afterTransmit, "10111");

detectadoErro = CRCUtil.detectErrorsCRC(crcRemainder);

} else if(errorMethod == 2) { **//Hamming**

toTransmit = HammingUtil.hamming(bitsDados);

afterTransmit = insertErrorsAuto(toTransmit, probErro);

detectadoErro = HammingUtil.hasError(afterTransmit);

hammingErrorPos = HammingUtil.countCheck(afterTransmit);

hammingErrorFix = HammingUtil.fixError(afterTransmit);

hammingOriginal = HammingUtil.getOriginal(hammingErrorFix);

}

}

**//desenhar o motor de demonstração**

protected void paintComponent(Graphics g) {

g.setColor(Color.BLACK);

Rectangle size = g.getClipBounds();

g.fillRect(0,0,size.width,size.height);

g.setColor(Color.GREEN);

g.setFont(new Font("Arial",Font.BOLD,12));

g.drawString("DEMONSTRAÇÃO", 10, 19);

String metd = "BIT PARIDADE";

if(errorMethod == 1)

metd = "CRC";

if(errorMethod == 2)

metd = "HAMMING";

g.drawString("ERROR DETECTION METHOD: "+metd, 10, 32);

g.drawString("#BITS: "+numBits, 10, 45);

if(manual){

g.drawString("MODE: MANUAL", 10, 58);

g.drawString("DATA: "+bitsDados, 10, 71);

g.drawString("ERROR PATTERN: "+padraoErros, 10, 84);

} else {

g.drawString("MODE: RANDOM", 10, 58);

g.drawString("DATA (GENERATED): "+bitsDados, 10, 71);

g.drawString("ERROR CHANCE: "+(int)(probErro\*100)+"%", 10, 84);

}

g.setColor(Color.RED);

g.drawString("----------------------------", 10, 97);

g.setColor(Color.GREEN);

g.drawString("SENT: "+toTransmit, 10, 110);

g.drawString("RCVD: "+afterTransmit, 10, 123);

if(detectadoErro){

g.setColor(Color.RED);

g.drawString("ERROR FOUND", 10, 136);

if(errorMethod == 2){

g.drawString("ERROR POS: "+hammingErrorPos, 10, 149);

g.drawString("FIX: "+hammingErrorFix, 10, 162);

}

g.setColor(Color.GREEN);

} else {

g.drawString("NO ERRORS FOUND", 10, 136);

}

if(errorMethod == 1){

g.drawString("REMAINDER: "+crcRemainder, 10, 149);

} else if (errorMethod == 2){

int pos = 149;

if(detectadoErro){

pos = 175;

}

g.drawString("FINAL MESSAGE: "+hammingOriginal, 10, pos);

}

}

}

* CRCUtil.java

public class CRCUtil {

**/\***

**Class CRCUtil, contém métodos que implementam a técnica de transmissão com CRC.**

**\*/**

**/\***

**Divisao de polinómio, tem como inputs o dividendo, divisor e o resto,**

**devolve um vector de inteiros com a divisao.**

**\*/**

private static int[] dividePol(int div[],int divisor[], int rem[]){

int cur=0;

while(true){

for(int i=0;i<divisor.length;i++)

rem[cur+i]=(rem[cur+i]^divisor[i]);

while(rem[cur]==0 && cur!=rem.length-1)

cur++;

if((rem.length-cur)<divisor.length)

break;

}

return rem;

}

**/\***

**método crcRem**

**A partir da mensagem com crc e o divisor, devolve uma String com o resto (remainder) da operação de CRC**

**\*/**

public static String crcRem(String crcStr, String divisorStr){

int[] crc = new int[crcStr.length()];

int[] rem = new int[crcStr.length()];

int[] divisor = new int[divisorStr.length()];

//convertemos a string em vectores de inteiros (1,0)

for(int i=0; i < crcStr.length(); i++){

if(crcStr.charAt(i) == '0'){

crc[i] = 0;

} else {

crc[i] = 1;

}

}

for(int i=0; i < divisorStr.length(); i++){

if(divisorStr.charAt(i) == '0'){

divisor[i] = 0;

} else {

divisor[i] = 1;

}

}

for(int j=0; j<crc.length; j++){

rem[j] = crc[j];

}

rem=dividePol(crc, divisor, rem);

String remStr = ""; //**Usamos isto para converter de volta para uma String com os dados.**

for(int i=0; i<rem.length;i++){

if(rem[i]==0){

remStr=remStr+'0';

} else {

remStr=remStr+'1';

}

}

return remStr;

}

**/\***

**método detectErrorsCRC**

**utilizado para detectar erros no código CRC através do Resto (remainder)**

**Após a operação, caso encontre um bit 1, retorna true (encontrou erro)**

**caso contrário retorna false, ou seja, a técnica não detectou nenhum erro na trama.**

**\*/**

public static boolean detectErrorsCRC(String remStr){

int[] rem = new int[remStr.length()];

for(int i=0; i < remStr.length(); i++){

if(remStr.charAt(i) == '0'){

rem[i] = 0;

} else {

rem[i] = 1;

}

}

for(int i=0; i < rem.length; i++){

if(rem[i]!=0){

return true;

}

if(i==rem.length-1)

return false;

}

return false;

}

**/\***

**método crc, este é o método principal desta classe,**

**tem como input a mensagem a ser codificada e o divisor original (que no caso das tramas de 4 e 11 bits neste projecto vai ser sempre 10111)**

**mas o método mantém-se dinâmico e pode ser utilizado para outros polinómios geradores.**

**Retorna a mensagem já codificada com CRC.**

**\*/**

public static String crc(String message, String divisorOrig){

int[] data = new int [message.length()];

int[] divisor = new int [divisorOrig.length()];

int total\_size = message.length() + divisorOrig.length() - 1;

**/\***

**Convertemos as Strings com os bits dados para vectores de inteiros, para facilitar**

**o processo.**

**\*/**

for(int i=0;i<message.length();i++){

if(message.charAt(i) == '0'){

data[i] = 0;

} else {

data[i] = 1;

}

}

for(int i=0;i<divisorOrig.length();i++){

if(divisorOrig.charAt(i) == '0'){

divisor[i] = 0;

} else {

divisor[i] = 1;

}

}

int[] div=new int[total\_size];

int[] rem=new int[total\_size];

int[] crc=new int[total\_size];

for(int i=0;i<data.length;i++)

div[i]=data[i];

for(int j=0; j<div.length; j++){

rem[j] = div[j];

}

rem=dividePol(div, divisor, rem);

for(int i=0;i<div.length;i++){

crc[i]=(div[i]^rem[i]);

}

**//converte-se o resultado de volta para String para trabalhar com ela nos motores.**

String str = "";

for(int i=0;i<crc.length;i++){

if(crc[i]==0){

str=str+'0';

} else {

str=str+'1';

}

}

return str;

}

}

* HammingUtil.java

public class HammingUtil {

**/\***

**Classe HammingUtil**

**contém os métodos que implementam a técnica de transmissão de dados com Códigos de Hamming**

**No caso do projecto, Hamming(7,4) e Hamming(15,11)**

**\*/**

**//Retorna true se '1', false se '0'**

private static boolean charToBool(char c){

if(c == '1')

return true;

return false;

}

**/\***

**Lista completa dos XORs necessarios para a implementação do código de hamming para mensagens de 4 e 11 bits**

**Os parity bits e os check bits foram calculados anteriormente**

**e apenas implementados nesta forma, sem ser generalizada.**

**Parity:**

**4bit**

**P1 = X3 + X5 + X7**

**P2 = X3 + X6 + X7**

**P4 = X5 + X6 + X7**

**11bit**

**P1 = X3 + X5 + X7 + X9 + X11 + X13 + X15**

**P2 = X3 + X6 + X7 + X10 + X11 + X14 + X15**

**P4 = X5 + X6 + X7 + X12 + X13 + X14 + X15**

**P8 = X9 + X10 + X11 + X12 + X13 + X14 + X15**

**Check:**

**4bit**

**C1 = X3 + X5 + X7 + P1**

**C2 = X3 + X6 + X7 + P2**

**C4 = X5 + X6 + X7 + P4**

**11bit**

**C1 = X3 + X5 + X7 + X9 + X11 + X13 + X15 + P1**

**C2 = X3 + X6 + X7 + X10 + X11 + X14 + X15 + P2**

**C4 = X5 + X6 + X7 + X12 + X13 + X14 + X15 + P4**

**C8 = X9 + X10 + X11 + X12 + X13 + X14 + X15 + P8**

**\*/**

**/\***

**método hamming, método principal que codifica a mensagem**

**Tem como input apenas a mensagem, toma partido do método XOR do MotorDemonstracao para realizar**

**as operações necessarias para os parity bits.**

**\*/**

public static String hamming(String codeStr){

int numBits = codeStr.length();

boolean[] message = new boolean[numBits];

for(int i=0;i<numBits;i++){

message[i] = charToBool(codeStr.charAt(i));

}

boolean[] finalCode = null;

if(numBits == 4){

boolean p1,p2,x3,p4,x5,x6,x7;

x3 = message[0];

x5 = message[1];

x6 = message[2];

x7 = message[3];

p1 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(x3, x5), x7);

p2 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(x3, x6), x7);

p4 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(x5, x6), x7);

boolean[] temp = {p1,p2,x3,p4,x5,x6,x7};

finalCode = temp;

} else if(numBits == 11){

boolean p1,p2,x3,p4,x5,x6,x7,p8,x9,x10,x11,x12,x13,x14,x15;

x3 = message[0];

x5 = message[1];

x6 = message[2];

x7 = message[3];

x9 = message[4];

x10 = message[5];

x11 = message[6];

x12 = message[7];

x13 = message[8];

x14 = message[9];

x15 = message[10];

p1 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x3,x5),x7),x9),x11),x13),x15);

p2 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x3,x6),x7),x10),x11),x14),x15);

p4 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x5,x6),x7),x12),x13),x14),x15);

p8 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x9,x10),x11),x12),x13),x14),x15);

boolean[] temp = {p1,p2,x3,p4,x5,x6,x7,p8,x9,x10,x11,x12,x13,x14,x15};

finalCode = temp;

}

String str = "";

for(int i = 0; i<finalCode.length;i++){

if(finalCode[i]){

str = str + "1";

} else {

str = str + "0";

}

}

return str;

}

**//método que retorna se a contagem de erros (ou seja a posição) é maior que 0, ou seja, se detecta um erro.**

public static boolean hasError(String codeStr){

return countCheck(codeStr) > 0;

}

**//método que substitui um caracter na posição index da String str por outro, rep**

private static String replaceCharString(String str, int index, char rep){

String fin = "";

for(int i = 0; i < str.length();i++){

if(i == index){

fin = fin + rep;

} else {

fin = fin + str.charAt(i);

}

}

return fin;

}

**/\***

**método getDistance**

**Retorna a distância de Hamming entre duas quaisquer Strings de bits de dados**

**Desde que estas tenham o mesmo tamanho.**

**Pode ser usada para calcular o número de bits errados entre uma mensagem e outra (como visto no simulador).**

**\*/**

public static int getDistance(String one, String two){

if(one.length() == two.length()){

boolean[] oneBool, twoBool, finalBool;

oneBool = new boolean[one.length()];

twoBool = new boolean[one.length()];

finalBool = new boolean[one.length()];

for(int i = 0; i<one.length();i++){

if(one.charAt(i) == '1'){

oneBool[i] = true;

} else {

oneBool[i] = false;

}

if(two.charAt(i) == '1'){

twoBool[i] = true;

} else {

twoBool[i] = false;

}

finalBool[i] = MotorDemonstracao.XOR(oneBool[i], twoBool[i]);

}

int dist = 0;

for(int i = 0; i<finalBool.length;i++){

if(finalBool[i])

dist++;

}

return dist;

}

return 0;

}

**/\***

**A partir da mensagem codificada com hamming, retorna a mensagem original**

**Ou seja, todas as que não são parity bits.**

**\*/**

public static String getOriginal(String codeStr){

int numBits = codeStr.length();

if(numBits == 7){

**//Hamming(7,4), retorna as posiçoes 2,4,5 e 6**

return ""+codeStr.charAt(2)+codeStr.charAt(4)+codeStr.charAt(5)+codeStr.charAt(6);

} else if(numBits == 15){

//**Hamming(15,11), retorna as posiçoes 2,4,5,6,8,9,10,11,12,13 e 14**

return ""+codeStr.charAt(2)+codeStr.charAt(4)+codeStr.charAt(5)+codeStr.charAt(6)+codeStr.charAt(8)+codeStr.charAt(9)+codeStr.charAt(10)+codeStr.charAt(11)+codeStr.charAt(12)+codeStr.charAt(13)+codeStr.charAt(14);

}

return codeStr;

}

/\*

**método fixError, que tem como função tentar corrigir uma trama dado que foi encontrado um erro.**

**Funciona por usar o método countCheck para descobrir se existe um erro, e a posição em que se encontra**

**e de seguida inverte o bit nessa posição.**

**Só é possivel, com este método, corrigir com 100% de certeza 1 erro de bit por trama.**

**\*/**

public static String fixError(String codeStr){

int errorPos = countCheck(codeStr);

//System.out.println("fixError - codeStr: "+codeStr+" - errorPos: "+errorPos+" char at errorPos: "+codeStr.charAt(errorPos-1));

if(errorPos > 0){

char err = codeStr.charAt(errorPos-1);

char fix = MotorDemonstracao.invertBinChar(err);

codeStr = replaceCharString(codeStr, errorPos-1, fix);

}

return codeStr;

}

**/\***

**método countCheck, utilizado em vários métodos acima**

**Toma partido dos checkbits para verificar a presença de erros na trama dada.**

**Utiliza o sistema listado no topo, com XORs entre as posições e o parity bit especifico.**

**Retorna a posição em que encontra (dado que encontra) um erro.**

**Consegue detectar com segurança até 2 erros. (apenas podendo ser corrigido 1)**

**\*/**

public static int countCheck(String codeStr){

int numBits = codeStr.length();

boolean[] message = new boolean[numBits];

for(int i=0;i<numBits;i++){

message[i] = charToBool(codeStr.charAt(i));

}

if(numBits == 7){

boolean p1,p2,x3,p4,x5,x6,x7,c1,c2,c4;

p1 = message[0];

p2 = message[1];

x3 = message[2];

p4 = message[3];

x5 = message[4];

x6 = message[5];

x7 = message[6];

c1 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x3, x5), x7),p1);

c2 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x3, x6), x7),p2);

c4 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x5, x6), x7),p4);

int tot = 0;

if(c1)

tot++;

if(c2)

tot+=2;

if(c4)

tot+=4;

return tot;

} else if(numBits == 15){

boolean p1,p2,x3,p4,x5,x6,x7,p8,x9,x10,x11,x12,x13,x14,x15,c1,c2,c4,c8;

p1 = message[0];

p2 = message[1];

x3 = message[2];

p4 = message[3];

x5 = message[4];

x6 = message[5];

x7 = message[6];

p8 = message[7];

x9 = message[8];

x10 = message[9];

x11 = message[10];

x12 = message[11];

x13 = message[12];

x14 = message[13];

x15 = message[14];

c1 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x3,x5),x7),x9),x11),x13),x15),p1);

c2 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x3,x6),x7),x10),x11),x14),x15),p2);

c4 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x5,x6),x7),x12),x13),x14),x15),p4);

c8 = MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(MotorDemonstracao.XOR(x9,x10),x11),x12),x13),x14),x15),p8);

int tot = 0;

if(c1)

tot++;

if(c2)

tot+=2;

if(c4)

tot+=4;

if(c8)

tot+=8;

return tot;

}

return 0;

}

}