Simulador de Algorítimos usados em Comunicação de Dados

Clarel de Menezes Spies Filho

Departamento de Informática – Universidade de Santa Cruz do Sul (UNISC)
96.815-900– Santa Cruz do Sul – RS – Brasil

clarel.filho@gmail.com

Resumo. Na comunicação de dados é necessário testes nas informações recebidas para garantir a integridade da informação que será processada. A ideia deste artigo é apresentar alguns destes algorítimos de verificação de erro e como foram implementado em um simulador desenvolvido.

Abstract. In data communication is needed tests in received data to garanty the integrity of the information that will be processed. The idea of the paper is to present some of those algoritmos of error verification and how they were implemented in one developed simulator.

1. INTRODUÇÃO

Na comunicação de dados com estudo baseado no modelo OSI, durante a transmissão das informações na camada um do modelo OSI (Camada Física) verificamos que pode ocorrer diferença entre o sinal enviado e o sinal recebido entre estações devido há interferência de sinal elétrico, por exemplo [1].

Dependendo do meio de como é feito esta comunicação a probabilidade destes erros ocorrerem podem ser maiores ou menos, mas de qualquer forma devemos nos assegurar de que a informação processada seja exatamente a informação que foi enviada. Quando um erro é detectado há diversas maneiras de prosseguir, pode ser sinalizados, corrigido ou ser feito um pedido de retransmissão da informação [1].

Para isto na camada dois do modelo OSI (Camada de Enlace) é implementado algoritmos de verificação da informação recebida, e em alguns casos, dependendo do meio de transmissão e da necessidade algorítimos de correção também. Para isto o transmissor e o receptor devem combinar qual será o algoritmo responsável por esta verificação e a escolha pode ser baseada no meio físico usado na transmissão e no tipo de serviço que será usado [2].

Assim então, o transmissor será responsável por fazer a codificação da mensagem, que seria basicamente adicionar informações de redundância para permitir o que o receptor

verifique se a mensagem recebida está correta e a corrija caso seja possível para depois decodificá-la [2].

Os serviços de codificação, verificação, correção de erro e decodificação da mensagem recebida poderão ser simulados no software que será apresentado, atualmente suporta os algorítimos de Código de Hamming, CRC (Cyclic Redundancy Check), Paridade Par e Paridade Ímpar.

2. SIMULADOR

O simulador foi desenvolvido na linguagem Java e por isto é multiplataformas, e para ser executado será necessário que o sistema tenha o Java JRE 1.7 ou mais novo instalado, e então dar dois cliques no arquivo para executá-lo.

Figura 1. Tela do simulador aberto

O software está em linguagem português e é fácil de usar, devemos apenas inserir no campo Mensagem, a informação que queremos enviar no formato ASCII, selecionar o algoritmo que gostaríamos de simular (Hamming, Crc, Paridade Par ou Paridade Ímpar) e

marcarmos se queremos simular um erro na transmissão da mensagem ou não. Caso tenha marcado para simular o erro, o software trocará um bit em posição aleatória nos bits recebidos.

Após entrar com as informações e preferências de simulação deve-se clicar em executar para obtermos os resultados que serão exibidos nos demais campos do software.

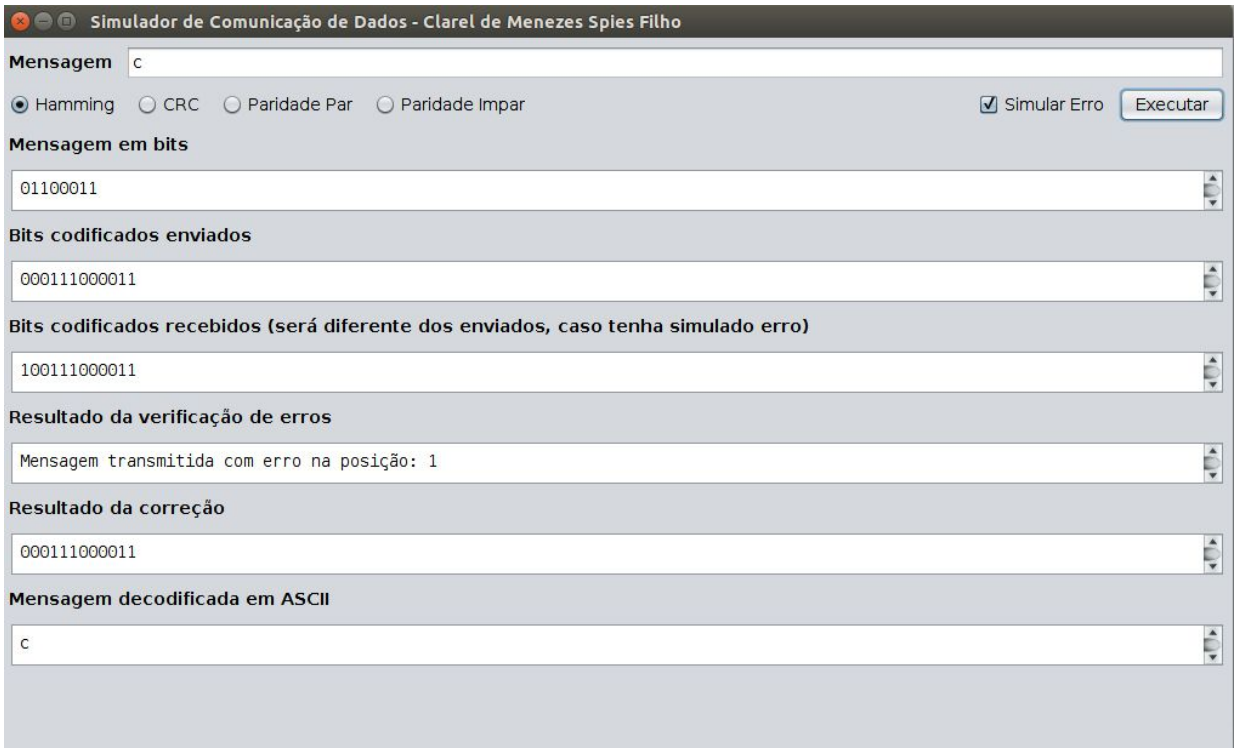

Figura 2. Tela do simulador após execução

No campo Mensagem em bits, será exibida a conversão da mensagem de ASCII para bits, cada caractere da mensagem serão representados por oito bits. No campo Bits codificados enviados é exibido a mensagem em formato bits juntamente com os dados de redundância gerados pelo algorítimo selecionado.receptor teria recebido após a transmissão pelo meio físico, neste campo caso você tenha selecionado para simular erro, haverá um bit aleatório trocado para provocar o erro na informação.

No campo Resultado da verificação de erros será exibido a mensagem de erro detectado, ou não. No caso de haver um erro no algorítimo de Hamming ele informará qual a posição do vetor de bits está o bit provavelmente errado. No campo resultado da correção será exibido o vetor recebido após a execução da correção para o algorítimo de Hamming, para os demais algorítimos não se aplica correção.

No campo Mensagem decodificado em ASCII será exibida a mensagem após a decodificação do vetor de bits recebido e corrigido no caso de Hamming. Observando esta informação podemos ver a diferença na mensagem que foi recebida no caso de haver algum erro. Em alguns casos o simulador gera o erro em um dos bits redundantes, o que acaba gerando um alerta de erro mas não influencia na mensagem recebida, mantendo a informação correta.

3. CÓDIGO DE HAMMING

Desenvolvido por Richard Hamming, este algoritmo permite a detecção e a correção de erros. Para codificar basicamente devemos reorganizar o vetor de bits a ser enviados, adicionando os bits redundantes na posição do vetor que são potência de dois, estes são os chamados bits de controle [3].

Exemplo de codificação baseado no caractere C, o resultado da conversão do caractere ASCII para bits forma o seguinte resultado:

01000011;

Reorganizando o vetor e adicionado os caracteres de redundância o vetor fica no forma:

P1 P2 D1 P3 D2 D3 D4 P4 D5 D6 D7 D8, onde o D representa um bit de dados e P representa um bit redundância.

Para calcularmos o valor dos bits de redundância, a forma utilizada foi pegar a posição do array onde está o bit que será calculado e realizar os passos a seguir sempre de maneira semelhante, apenas mudando os bits que deve-se pegar para o cálculo conforme a posição de controle que será calculada.

P1: Ignoramos a posição atual e pula a próxima, no caso P2, pegamos D1, pulamos P3, pegamos D2, pulamos D3, pegamos D4, pulamos P4 e assim por diante até o final do vetor.

P2: ignoramos a posição atual e pegamos o D1, pulamos P3 e D2, pegamos D3 e D4, pulamos P4 e D5 e assim por diante até o final do vetor.

P3: Ignoramos a posição atual, pegamos D2 e D3, pulamos D4 P4 e D5, pegamos D6 D7 e D8.

Com os bits obtidos em cada um dos cálculos de controle, executamos a operação XOR e o seu resultado será o valor do bit de controle, como no exemplo abaixo:

P1: 0 + 0 + 0 + 1 = 0;

A mensagem depois de todos os cálculos ficará codificada na seguinte forma, e será enviada ao transmissor.

P0 P1 D0 P0 D1 D0 D0 P0 D0 D0 D1 D1;

```
public StringBuilder encode(String asciiMessage) {
    StringBuilder mensagemRecebidaEmBinario = Utilidades.converteAsciiParaBinario(asciiMessage);
    int messageLenght = mensagemRecebidaEmBinario.toString().length();
    StringBuilder mensagemCodificada = new StringBuilder();
    int index = 0;
    while (index < messageLenght) {
        while (Utilidades.potenciaDeDois(mensagemCodificada.toString().length() + 1)) {
            //coloca 0 para evitar valores null, depois será cálculado o real valor
            mensagemCodificada.append("0");
        }
        mensagemCodificada.append(mensagemRecebidaEmBinario.charAt(index));
        index++;
    }
    messageLenght = mensagemCodificada.toString().length();
    index = 0;
    while (index < messageLenght) {
        if (Utilidades.potenciaDeDois(index + 1)) {
            StringBuilder bitsParaExecutarXor = new StringBuilder();
            int auxIndex = index;
            //usado para contar, usa 2 bits, pula, usa 2, assim por diante.. depende do index
            boolean indexIsProductOfSum = true;
            int totalAux = 0;
            while (auxIndex < messageLenght) {
                totalAux++;
                //para não permitir adicionar o valor na posição index na operação xor
                if (indexIsProductOfSum && auxIndex != index) {
                    bitsParaExecutarXor.append(mensagemCodificada.charAt(auxIndex));
                }
                if (totalAux == index + 1) {
                    indexIsProductOfSum = !indexIsProductOfSum;
                    totalAux = 0;
                }
                auxIndex++;
            }
            mensagemCodificada.setCharAt(index, Utilidades.executaXorEmArrayBinario(bitsParaExecutarXor).charAt(0));
        }
        index++;
    }
    return mensagemCodificada;
}
```

Figura 3. Algorítimo de codificação Hamming

Para verificar a existência de erros em cada bit de controle, respectivamente a sua posição vamos intercalando e guardando estes bits. Por exemplo o bit da posição um, pegamos ele, ignora um, pega um, ignora um assim por diante até o fim do vetor. Depois executar o XOR nestes valores obtidos para cada bit de controle, depois olhamos o vetor obtido por todos os XOR calculados em cada bit de controle. Invertermos estes bits calculados e descobrimos o valor dele, que será um inteiro que revelará a posição do bit com valor errado. Se o valor obtido for igual a zero, não há erro [3].

```java
public String verificaErro(StringBuilder mensagemCodificada) {
    int messageLenght = mensagemCodificada.toString().length();
    StringBuilder seguraBinariosVerificacao = new StringBuilder();
    int index = 0;
    while (index < messageLenght) {
        if (Utilidades.potenciaDeDois(index + 1)) {
            StringBuilder bitsParaExecutarXor = new StringBuilder();
            int auxIndex = index;
            //usado para contar, usa 2 bits, pula 2.. depende do valor do index.
            boolean indexProdutoDaSoma = true;
            int totalAux = 0;
            while (auxIndex < messageLenght) {
                totalAux++;
                if (indexProdutoDaSoma) {
                    bitsParaExecutarXor.append(mensagemCodificada.charAt(auxIndex));
                }
                if (totalAux == index + 1) {
                    indexProdutoDaSoma = !indexProdutoDaSoma;
                    totalAux = 0;
                }
                auxIndex++;
            }
            seguraBinariosVerificacao.append(Utilidades.executaXorEmArrayBinario(bitsParaExecutarXor));
        }
        index++;
    }
    seguraBinariosVerificacao = seguraBinariosVerificacao.reverse();
    this.posicaoErro = Integer.parseInt(seguraBinariosVerificacao.toString(), 2);
    if (posicaoErro > 0) {
        return "Mensagem transmitida com erro na posição: " + this.posicaoErro;
    } else {
        return "Mensagem transmitida sem erro";
    }
}
```

Figura 4. Algorítimo de verificação de erro Hamming

Para corrigir o erro devemos inverter o valor do bit na posição de valor encontrado pelo método de verificação de erro [3].

```java
public StringBuilder corrigeErro(StringBuilder mensagemCodificada) {
    return Utilidades.mudaUmValorDeterminadoNoArrayDeBits(mensagemCodificada, this.posicaoErro - 1);
}
```

Figura 5. Algorítimo de correção de erro Hamming

Para realizar a decodificação e obtermos os dados que foram enviados, apenas devemos retirar do vetor de bits as informações que estão na posição potência de dois, voltando assim para a informação original [3].

```java
public String decode(StringBuilder mensagemEmBinario) {
    int messageLenght = mensagemEmBinario.length();
    StringBuilder mensagemDecodificada = new StringBuilder();
    int index = 0;
    while (index < messageLenght) {
        if (!Utilidades.potenciaDeDois(index + 1)) {
            mensagemDecodificada.append(mensagemEmBinario.charAt(index));
        }
        index++;
    }
    return Utilidades.converteBinarioParaAscii(mensagemDecodificada);
}
```

Figura 6. Algorítimo de decodificação Hamming

4. CRC

Neste algorítimo o transmissor e o receptor combinam qual será o polinômio, que será um vetor de bits usado como divisor no cálculo. Os bits da mensagem serão acrescidos de zeros, o número de zeros será o tamanho do polinômio menos um e estes bits serão o dividendo do cálculo. Após termos estas informações realizamos a divisão, e o resto dessa divisão vamos pegar os bits mais significativos de tamanho do polinômio e concatenaremos a mensagem original, este resto é o chamado CRC. Este algorítimo não implementa correção [3].

```java
@Override
public StringBuilder encode(String mensagemAscii) {
    StringBuilder mensagemEmBinario = Utilidades.converteAsciiParaBinario(mensagemAscii);
    String auxiliar = StringUtils.rightPad("", polinomio.length() - 1, "0");

    StringBuilder restoDivisao = Utilidades.retornaRestoDivisaoBits(new StringBuilder(mensagemEmBinario + auxiliar),
            polinomio);

    return mensagemEmBinario.append(restoDivisao.substring(1, polinomio.length()));
}

@Override
public String decode(StringBuilder mensagemEmBinario) {
    StringBuilder array = new StringBuilder(
            mensagemEmBinario.subSequence(0, mensagemEmBinario.length() - (polinomio.length() - 1)));

    return Utilidades.converteBinarioParaAscii(array);

}
```

Figura 7. Algorítimo de codificação CRC

Para verificar se há erro na mensagem recebida decodificada, dividimos ela pelo polinômio, e se o valor do resto for diferente de zero, há erro, do contrário a mensagem está correta [3].

```java
@Override
public String verificaErro(StringBuilder mensagemCodificada) {
    StringBuilder restoDivisao = Utilidades.retornaRestoDivisaoBits(new StringBuilder(mensagemCodificada), polinomio);
    int resultadoRestoDivisao = Integer.valueOf(restoDivisao.toString());

    if (resultadoRestoDivisao != 0) {
        return "Mensagem transmitida com erro";
    } else {
        return "Mensagem transmitida sem erro";
    }

}
```

Figura 8. Algorítimo de verificação de erro CRC

Para realizarmos a decodificação e obtermos os dados que foram enviados apenas devemos remover do vetor de bits codificados os bits mais significativos do tamanho do grau do polinômio, que é o tamanho do polinômio menos um [3].

```java
@Override
public String decode(StringBuilder mensagemEmBinario) {
    StringBuilder array = new StringBuilder(
            mensagemEmBinario.subSequence(0, mensagemEmBinario.length() - (polinomio.length() - 1)));

    return Utilidades.converteBinarioParaAscii(array);

}
```

Figura 9. Algorítimo de decodificação CRC

5. PARIDADE PAR OU ÍMPAR

Entre os algorítimos implementados este é o mais simples e o menos confiável, recomendado o uso apenas para transmissão de pequenas informações pois a contagem de 1's pares ou ímpares faz deste algorítimo muito simples e pouco confiável para mensagens longas [3].

Para realizar a codificação devemos contar o números de 1's no vetor de bits, e então adicionar mais um bit no final do vetor, que será o bit de redundância. Se a paridade escolhida for par, com a contagem de 1's mais este bits deve haver um número par de 1's [3].

Para a paridade ímpar é feito o mesmo processo, mas no final o número de 1's na mensagem codificada deve ser um número ímpar [3].

Esta codificação apenas vai alterar a mensagem adicionado um bit no final dela [3].

```java
@Override
public StringBuilder encode(String asciiMessage) {
    StringBuilder binaryMessage = Utilidades.converteAsciiParaBinario(asciiMessage);

    if (Utilidades.contaUmEmArrayBinario(binaryMessage) % 2 == 0) {
        binaryMessage.append("1");
    } else {
        binaryMessage.append("0");
    }
    return binaryMessage;
}
```

Figura 10. Algorítimo de codificação Paridade Ímpar

```java
@Override
public StringBuilder encode(String asciiMessage) {
    StringBuilder binaryMessage = Utilidades.converteAsciiParaBinario(asciiMessage);

    if (Utilidades.contaUmEmArrayBinario(binaryMessage) % 2 != 0) {
        binaryMessage.append("1");
    } else {
        binaryMessage.append("0");
    }
    return binaryMessage;
```

Figura 11. Algorítimo de codificação Paridade Par

Para realizar a verificação de erros na mensagem recebida, contamos o número de 1's, se o número encontrado for um número par a mensagem foi transmitida com sucesso no algorítimo de paridade par, e errada para algorítimo de paridade ímpar [3].

Se o número de 1's encontrados for um número ímpar, a mensagem foi transmitida com sucesso para algorítimo de paridade ímpar, e errada para o algorítimo de paridade par [3].

```java
@Override
public String verificaErro(StringBuilder codifiedMessage) {
    if (Utilidades.contaUmEmArrayBinario(codifiedMessage) % 2 == 0) {
        return "Mensagem transmitida com erro";
    } else {
        return "Mensagem transmitida sem erro";
    }

}
```

Figura 12. Algorítimo de verificação de erro Paridade Ímpar

```java
@Override
public String verificaErro(StringBuilder codifiedMessage) {
    if (Utilidades.contaUmEmArrayBinario(codifiedMessage) % 2 != 0) {
        return "Mensagem transmitida com erro";
    } else {
        return "Mensagem transmitida sem erro";
    }

}
```

Figura 13. Algorítimo de verificação de erro Paridade Par

Para realizarmos a decodificação da mensagem recebida, apenas retiramos o último bit do vetor, pois ele é o bit de redundância. Para este algorítimo não há correção de erro [3].

```java
@Override
public String decode(StringBuilder binaryMessage) {
    StringBuilder decodedMessage = new StringBuilder(binaryMessage.substring(0, binaryMessage.toString().length()-1));

    return Utilidades.converteBinarioParaAscii(decodedMessage);
}
```

Figura 14. Algorítimo de decodificação para paridade ímpar e par

6. CONSIDERAÇÕES FINAIS

Com pesquisa e estudo realizado foi possível ter uma visão de como é feito a transmissão dos dados, bem como da importância dos algorítimos de detecção e correção. Seria impossível, por exemplo, ter um sistema de banco transmitindo dados errados, sem nenhuma verificação na mensagem transmitida.

Espero que o simulador seja usado por pessoas que queiram aprender o funcionamento destes algorítimos e realizar testes para aprofundar o seu conhecimento. Serão implementadas funções como informar um polinômio desejado, e aceitar mensagem no formato binário para iniciar transferências. Acredito que isto irá aumentar um potencial público alvo do software.

O referido software está disponível no Github no endereço https://github.com/clarel/SimuladorComunicacaoDeDados.

Referências

[1] “Redes de computadores e a Internet”, https://www.ic.unicamp.br/~ripolito/peds/st564/material/Camada_de_Enlace-1.pdf, acesso em 03 dezembro 2015.

[2]“A camada de Enlace”, https://docente.ifrn.edu.br/tadeuferreira/disciplinas/2012.1/redes-i-eja/Aula12.pdf, acesso em 03 dezembro 2015.

[3]“NTFS”, http://www.ece.ufrgs.br/~fetter/ele00012/enlace_dados.pdf, acesso em 03 dezembro 2015.