UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ - UNIOESTE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Comparativo da Eficácia dos Mecanismos de Detecção de Erros da Pilha de Protocolos da Internet Utilizando Simulação

Trabalho de Conclusão de Curso

Igor França Negrizoli

Cascavel–PR

2021

UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ - UNIOESTE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

Colegiado de Ciência da Computação

Curso de Bacharelado em Ciência da Computação

Igor França Negrizoli

Comparativo da Eficácia dos Mecanismos de Detecção de Erros da Pilha de Protocolos da Internet Utilizando Simulação

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel.

Orientador(a): Prof. Dr. Luiz Antonio Rodrigues

Cascavel–PR

2021

IGOR FRANÇA NEGRIZOLI

COMPARATIVO DA EFICÁCIA DOS MECANISMOS DE DETECÇÃO DE ERROS DA PILHA DE PROTOCOLOS DA INTERNET UTILIZANDO SIMULAÇÃO

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Dr. Luiz Antonio Rodrigues (Orientador)
Colegiado de Ciência da Computação, UNIOESTE

Prof. Dr. Guilherme Galante
Colegiado de Ciência da Computação, UNIOESTE

Prof. Dr. Marcio Seiji Oyamada
Colegiado de Ciência da Computação, UNIOESTE

Cascavel, 29 de julho de 2022.

Este trabalho é dedicado às crianças curiosas que
se encantaram com uma tela colorida, teclado e mouse
e decidiram que um dia descobririam todos os segredos
dessa máquina misteriosa.

Agradecimentos

Agradeço à minha família e amigos a todo apoio que me deram durante todos os anos da minha graduação e a todos os professores, desde os do ensino superior até os do ensino básico, por me conduzirem até aqui, mas principalmente à minha primeira professora, minha mãe Eluiza por todo incentivo e apoio dado durante toda minha vida. Agradeço também ao grupo PETComp por incentivar minha permanência no curso e meu bom desempenho na graduação. E também ao Casimiro por me arrancar várias risadas nos poucos momentos de descanso que tive durante o ano.

Eu disparo e paro no infinito
Reabasteço, sigo em frente, é bonito
Viajo pelo espaço e, o que eu vejo, eu deixo escrito
E só Jah Jah pode me dar um veredito
- Black Alien

Resumo

Este trabalho apresenta uma análise dos mecanismos de detecção de erros de transmissão mais utilizados em redes TCP/IP com atenção especial às suas vulnerabilidades, bem como o estudo da viabilidade de modelos de erros usados para simular erros de transmissão de forma artificial. O objetivo é avaliar a eficácia dos mecanismos por meio de injeção de erros no campo de dados da aplicação utilizando um simulador de envio de pacotes. Este simulador produz pacotes com conteúdo gerado aleatoriamente e simula erros de transmissão. Com essas operações ele é capaz de coletar dados estatísticos sobre a eficácia dos principais mecanismos de tolerância a falhas em redes utilizados atualmente em redes TCP/IP. Este simulador se encontrava parcialmente implementado e possuía duas ferramentas de detecção de erros, o *Checksum* e o CRC, e também três modelos de injeção de erros, os modelos de Bernoulli, Gilbert e Periódico. Para complementar o simulador, foram implementados os algoritmos de detecção de erros de Fletcher e Adler e um novo modelo de injeção de erros proposto pelo autor. Os resultados obtidos indicam que existem cenários específicos em que cada algoritmo deve ser utilizado. Discussões acerca das características de cada cenário e os comportamentos apresentados pelos algoritmos foram realizadas. Para isso foram feitos testes utilizando o simulador desenvolvido que mediam eficácia dos algoritmos de detecção de erros e seu tempo de execução.

Palavras-chave: Redes de computadores. Tolerância à falhas. Ambiente de simulação. Injeção de erros.

Lista de figuras

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

BER	*Bit Error Rate*
BOR	*Burst Occurrence Rate*
CRC	*Cyclic Redundancy Check*
ICMP	*Internet Control Message Protocol*
IoT	Internet das Coisas
IP	*Internet Protocol*
IPv4	*Internet Protocol Version 4*
SSL	*Secure Sockets Layer*
TCP	*Transmission Control Protocol*
TLS	*Transport Layer Security*
UDP	*User Datagram Protocol*
CCITT	Comitê Consultivo de Telefonia e Telegrafia Internacional

Lista de símbolos

$G(x)$	Polinômio gerador
c_1	Primeira soma do algoritmo de Fletcher e Adler.
c_2	Segunda soma do algoritmo de Fletcher e Adler.
k	Tamanho dos blocos do pacote de dados.
K	Tamanho das rajadas com maior frequência produzidas pelo modelo de Gilbert.
p	Probabilidade de transição do estado *non-loss* para o estado *loss*.
q	Probabilidade de transição do estado *loss* para o estado *non-loss*.
e	Taxa com que ocorrem os erros no Modelo de Gilbert.
F_{Min}	Comprimento mínimo da rajada de erros no Modelo Periódico.
F_{Max}	Comprimento máximo da rajada de erros no Modelo Periódico.
T_{Min}	Intervalo mínimo entre o início das rajadas no Modelo Periódico.
T_{Max}	Intervalo máximo entre o início das rajadas no Modelo Periódico.
N	Tamanho do pacote de dados.
F	Tamanho das rajadas de erros injetadas no Modelo de Rajadas Esparsas.

Sumário

1 Introdução

A transferência de dados em redes de computadores está sujeita a diversos erros de transmissão, que podem ser causados por interferências eletromagnéticas ou falhas de componentes, tanto de hardware, quanto de software, gerando inversão ou inserção/supressão de bits. Além disso, a distribuição dos erros não é homogênea, podendo ocorrer em um único bit ou em rajadas, com vários bits consecutivos afetados (MAXINO; KOOPMAN, 2009; AZAHARI et al., 2014; KOOPMAN; DRISCOLL; HALL, 2015). Segundo Kurose e Ross (2012), o desafio de proporcionar uma transferência de dados confiável não existe apenas no contexto da camada de transporte, mas também nas camadas de enlace e de aplicação. Para superar o problema da transferência de dados confiável, três aspectos devem ser levados em consideração: detecção do erro, *feedback* do receptor, retransmissão.

A pilha de protocolos da Internet, também conhecida como TCP/IP, dispõe de mecanismos de detecção de erros implementados em diferentes camadas. Nas camadas de Transporte e Rede, a soma de verificação (do inglês, *Checksum*) é o padrão dos protocolos TCP (*Transmission Control Protocol*), UDP (*User Datagram Protocol*), IP (*Internet Protocol*) e ICMP (*Internet Control Message Protocol*). Na camada de Enlace, o CRC (do inglês, *Cyclic Redundancy Check*) é utilizado pelos protocolos Ethernet (IEEE 802.3) e Wi-Fi (IEEE 802.11), entre outros (KUROSE; ROSS, 2012). Em relação ao escopo, o CRC aplicado na camada de enlace tem a responsabilidade de verificar as transmissões ponto-a-ponto, recalculando o código de detecção de erros a cada repasse do pacote. O mesmo ocorre com os pacotes IPv4, visto que os campos do cabeçalho podem ser modificados em cada salto e a soma deve ser refeita. O *Checksum*, é utilizado na comunicação fim-a-fim, sendo calculado e verificado somente pelos processos da aplicação (emissor e receptor). Versões aprimoradas do *Checksum*, o *Checksum* de Fletcher (FLETCHER, 1982) e o de Adler (DEUTSCH, 1996), são amplamente discutidas como alternativas ao seu antecessor. O algoritmo de Fletcher já foi cogitado como algoritmo de detecção de erros para o protocolo TCP (ZWEIG; PARTRIDGE, 1990) e o algoritmo de Adler é usado atualmente na biblioteca de compressão de arquivos zlib.

Embora amplamente utilizados, esses algoritmos de detecção de erros possuem limitações em relação à sua eficácia (WOLF; BLAKENEY, 1988; STONE et al., 1998). O Checksum, por exemplo, usa códigos de verificação de 16 bits e detecta desde modificações unitárias de bits a rajadas de até 16 bits. O CRC, por sua vez, implementa um mecanismo mais robusto, normalmente com códigos de 32 bits, e capacidade de detecção de erros de bit únicos até rajadas de erros de até N bits, com alta probabilidade de detecção para rajadas maiores. No entanto, combinações de erros, especialmente em grandes rajadas, e falhas de verificação, podem passar despercebidos e alcançarem a aplicação (PANDURANGAN, 2016). O algoritmo de Fletcher é conhecido por ter potencial de detecção de erros comparável ao CRC e, mesmo assim, ter custo computacional menor. O algoritmo de Adler, por sua vez, faz uso de um número primo como seu divisor em módulo na tentativa de fazer com que os padrões de erros sejam detectados com probabilidade similar, porém isso o torna mais custoso (MAXINO; KOOPMAN, 2009).

Existem aplicações que possuem mecanismos eficientes de integridade embutidos no protocolo de sessão como a aplicação de códigos de *hash* (KUROSE; ROSS, 2012). No entanto, esta abordagem inclui o uso de algoritmos complexos de criptografia, que são custosos e desnecessários para as aplicações que não necessitam de garantias de confidencialidade e autenticidade, por exemplo. Além disso, o uso de tais mecanismos pode ser inviável em sistemas embarcados e de IoT (Internet das Coisas), pelo baixo poder de processamento destes dispositivos e aumento do consumo de energia. Isso faz relevante que seja feita também a analise do custo computacional necessário para executar vários cômputos desses algoritmos de detecção de erros para assim melhor avaliar a adequação de cada um deles a diferentes cenários.

1.1 Objetivos

O objetivo deste trabalho é apresentar uma análise dos mecanismos de detecção de erros implementados na pilha de protocolos da Internet por meio da utilização de estratégias de injeção de erros e simulação de modelos estatísticos que representam falhas de transmissão que podem ocorrer em diferentes camadas. Os resultados obtidos por Negrizoli, Rodrigues e Oyamada (2021) confirmam que erros podem passar despercebidos pelos diferentes mecanismos de verificação e alcançarem a camada de aplicação, podendo causar inconsistências e prejuízos aos usuários. Este trabalho apresenta uma análise da eficácia destes algoritmos em detectar erros de transmissão, assim como medições aproximadas do seu custo computacional complementando o software de simulação de envio de pacotes e as análises dos resultados feitas por Negrizoli, Rodrigues e Oyamada (2021).

Mais especificamente, avalia-se os algoritmos de verificação de erros que são comumente usados: o Checksum, o CRC, o algoritmo de Fletcher e o algoritmo de Adler. Para cada um destes algoritmos, é avaliado as implementações que geram tanto palavras de 32 bits quanto palavras de 16 bits. Para as implementações do algoritmo de CRC, o polinômio pode possuir qualquer valor

representável com N bits, sendo N o tamanho da palavra gerada pelo algoritmo. No Capítulo 3 são descritos os polinômios escolhidos assim como os fatores que motivam a escolha de tais polinômios.

1.2 Metodologia

O software de simulação citado serve de base para o desenvolvimento deste trabalho. O simulador foi desenvolvido utilizando a linguagem C++ e teve sua estrutura foi significativamente alterada, porém a lógica seguida para o teste de eficácia dos algoritmos é a mesma. Além dos algoritmos de detecção citados anteriormente, também foi acrescido ao simulador um novo modelo de injeção de erros proposto pelo autor, estes algoritmos de detecção de erros tão bem quanto os modelos de injeção estão descritos no Capítulo 2. Para avaliar o custo computacional dos algoritmos foram desenvolvidas também rotinas de testes que medem o tempo de execução de várias iterações dos algoritmos disponíveis no simulador. Além disso, para melhor avaliar o funcionamento de combinações de dois algoritmos de detecção de erros, encontra-se disponível uma rotina de testes que simula o funcionamento de dois algoritmos de detecção de erros implementados em camadas diferentes trabalhando em conjunto. O funcionamento das rotinas de teste presentes no simulador se encontram descritas no Capítulo 3. O desempenho destes algoritmos de detecção de erros nestas diferentes rotinas de testes bem como discussões a respeito do desempenho destes algoritmos se encontram no Capítulo 4.

2

Referencial Bibliográfico

Neste capítulo são apresentados e discutidos os algoritmos de detecção de erros estudados neste trabalho tão bem quanto suas possíveis vulnerabilidades, e os modelos utilizados para simulação da ocorrência de erros em transmissão de pacotes em redes de computadores e suas principais características.

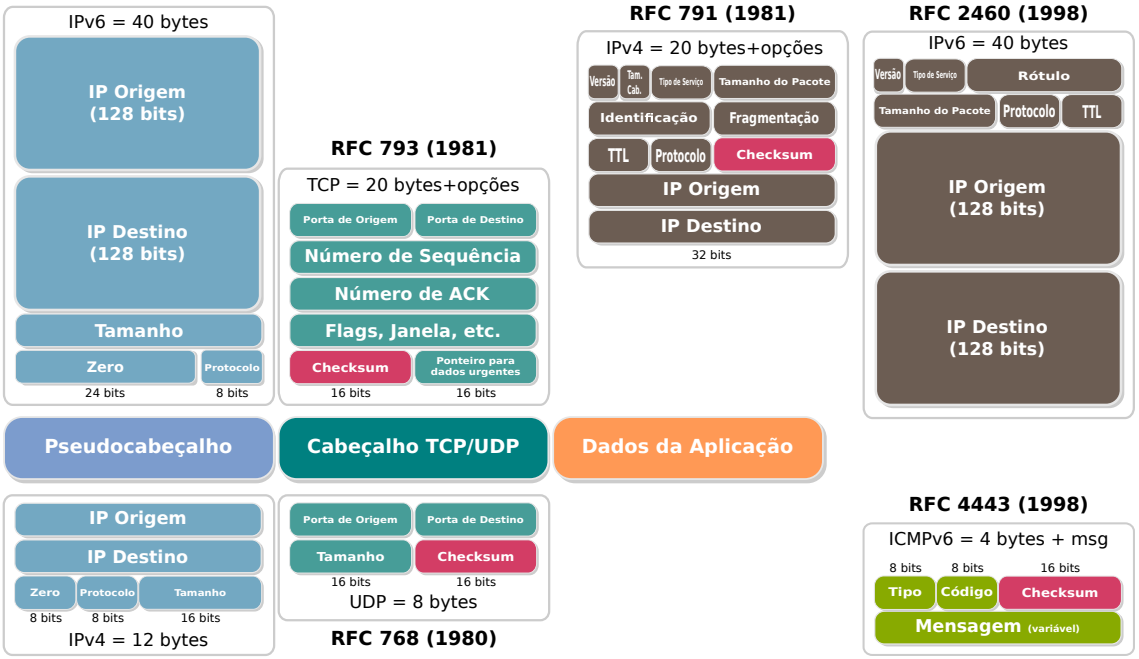
2.1 Detecção de erros de transmissão

De acordo com Plummer (1988) mensagens com erros não detectados que alcançam as camadas superiores podem acabar sendo ignoradas, por não fazerem sentido no contexto do protocolo. No caso do TCP, por exemplo, erros de transmissão podem ocasionar a interrupção da conexão, o que, embora inconveniente, não atinge a integridade dos dados. Por outro lado, erros não detectados que atingem a camada de aplicação podem ser danosos, especialmente se a aplicação não possui um mecanismo próprio de verificação.

Para detectar erros de transmissão, diversos protocolos empregam mecanismos que produzem uma palavra de tamanho definido pelo protocolo como é exposto pela Figura 1. Esta palavra é computada de acordo com regras bem definidas pelo protocolo e seu valor tem relação direta com os dados presentes no pacote. Sendo assim, esta palavra pode ser agregada ao cabeçalho do pacote pelo dispositivo transmissor, permitindo que o dispositivo receptor compute novamente a palavra de acordo com as regras definidas pelo protocolo e os dados recebidos. Ao comparar o resultado obtido com o que foi agregado ao cabeçalho do pacote é possível que o dispositivo receptor detecte erros de transmissão e descarte o pacote com dados inconsistentes.

As duas principais técnicas de detecção de erros utilizadas na pilha de protocolos TCP/IP são o *Checksum* e o CRC. Como alternativas a esses algoritmos de detecção de erros são considerados também o *Checksum* de Adler e Fletcher, que mesmo que não sejam utilizados na pilha de protocolos da internet, são úteis para outras aplicações como o Transport Protocol

Figura 1 – Cabeçalhos definidos para diferentes protocolos de transmissão



Fonte: Produzido pelo autor

(MCCOY, 1987) que têm o algoritmo de Fletcher como uma alternativa para computar sua palavra de verificação, e o formato Gzip (DEUTSCH; GAILLY, 1996) que utiliza o algoritmo de Adler para o cálculo da palavra de verificação. Esses algoritmos vêm sendo utilizados como principais mecanismos de detecção de erros por protocolos de transmissão nas últimas duas décadas, isso torna valioso o estudo de suas capacidades de detecção de erros. Os passos destes algoritmos citados estão descritos a seguir.

2.1.1 Checksum

Este algoritmo de detecção de erros é implementado nos protocolos TCP e UDP (camada de transporte) e IPv4 (camada de rede). Estes protocolos possuem um campo reservado ao *Checksum* que é enviado junto com os demais dados de identificação e controle. O cômputo do deste algoritmo de verificação está definido em Plummer (1988) para os protocolos TCP, UDP e IP:

1. Os bytes adjacentes são pareados para formar inteiros de 16 bits (considerando a implementação de 16 bits do *Checksum*) e a soma em nível binário é realizada em cada bloco. Caso ocorra o bit de *carry* na soma do bit mais significativo, uma unidade é adicionada ao resultado no bit menos significativo. Nesses protocolos (TCP, UDP e IPv4) o tamanho da chave de verificação é de 16 bits porém o conceito base do algoritmo permite implementações com tamanhos diferentes de código de verificação. Em uma implementação para um

Checksum com código de 32 bits, os bytes adjacentes são pareados formando inteiros de 32 bytes;

2. O campo em que está armazenado o *Checksum* não é considerado no cálculo e o complemento de um da soma (resultado binário da soma com os bits invertidos) é incluída no cabeçalho do protocolo;

3. A verificação do *Checksum* é feita pelo cálculo dos dados recebidos, incluindo o campo de *Checksum*, que deve resultar em todos os bits com valor 1 (um) para indicar uma transmissão possivelmente sem erros.

Em termos de eficácia, o *Checksum* detecta todos os erros de bit único, mas pode falhar para pares de erros de bit único alinhados verticalmente nas situações em que o valor dos valores dos pares diferem, visto que $0+1 = 1+0$ (NEGRIZOLI; RODRIGUES; OYAMADA, 2021). Esse cenário de falha não se aplica em situações em que os valores são iguais visto que $0+0 \neq 1+1$, mesmo que nesta posição de bit o resultado continue sendo 0 o erro é detectado devido à ativação do bit de *carry* (Figura 2) (AZAHARI et al., 2014). Este tipo de vulnerabilidade corrobora com a falha descrita por Stone et al. (1998) de que alterar a ordem da soma dos blocos não altera o resultado final do *Checksum* devido à propriedade comutativa da soma. Além disso, de acordo com Plummer (1988), mesmo que o cálculo do *Checksum* no receptor indique uma transmissão correta, a mensagem ainda pode conter erros com uma probabilidade de 2^{-d}, onde d é a quantidade de bits do *Checksum*. Portanto, em tese, o *Checksum* de 16 bits possui probabilidade de falha de $2^{-16} \approx 0,0152 * 10^{-3}$ e o *Checksum* de 32 bits possui $2^{-32} \approx 0,2328 * 10^{-9}$. Vale ressaltar que esta é uma estimativa de eficácia geral do protocolo e que ela não diz respeito a cenários em que padrões de erros específicos como os gerados pelo simulador ocorrem.

Figura 2 – Exemplo de alinhamento de erros não detectado pelo *Checksum*

Cálculo no envio							
1	0	1	0	**0**	1	0	1
0	1	0	0	**1**	0	1	0
1	1	1	0	1	1	1	1

→

Cálculo no recebimento							
1	0	1	0	**1**	1	0	1
0	1	0	0	**0**	0	1	0
1	1	1	0	1	1	1	1

Fonte: Produzido pelo autor

2.1.2 CRC

O CRC (*Cyclic Redundancy Check*) (PETERSON; BROWN, 1961) utiliza códigos polinomiais para detectar erros de transmissão, especialmente os erros em bits consecutivos (rajadas). É o mecanismo utilizado pelos protocolos de enlace, tanto em redes cabeadas, como o IEEE 802.3 (Ethernet), quanto sem fio e móveis, como o IEEE 802.11 (Wi-Fi), Bluetooth, CDMA e LoRa. O processo realizado é o seguinte:

1. Emissor e receptor definem um polinômio gerador $G(x)$ com grau máximo n. Quanto maior for o seu grau, maior será a capacidade de detecção de erros. Este polinômio é fixo durante toda a troca de mensagens do protocolo;

2. Os blocos de dados são concatenados a um conjunto de zeros equivalente ao grau do polinômio e divididos por $G(x)$; a divisão é feita com operações binárias de ou-exclusivo (XOR);

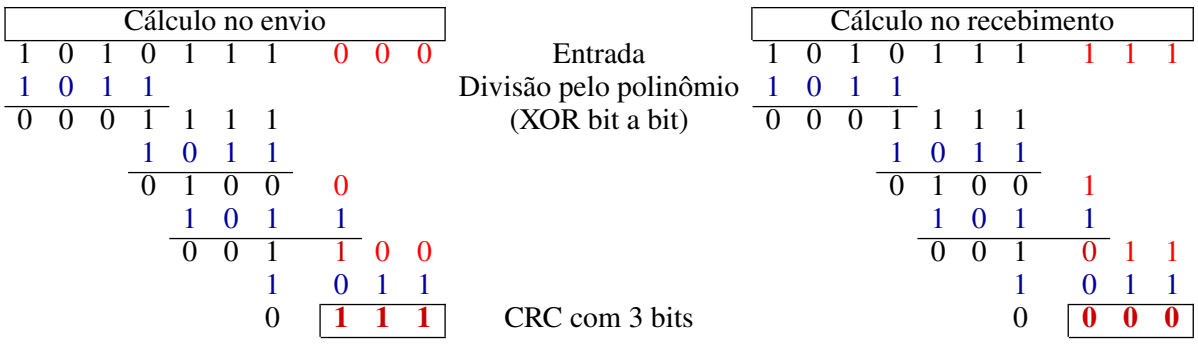
3. O resto da divisão, em quantidade de bits equivalente ao grau do polinômio, é utilizado como CRC;

4. Para verificar a integridade dos dados recebidos, o receptor anexa ao dado original o próprio CRC recebido e efetua a divisão pelo polinômio acordado. Um erro é detectado se o resto da divisão for diferente de zero.

Considere, por exemplo, o polinômio de grau três $x^3 + x + 1$, que pode ser representado por $1x^3 + 0x^2 + 1x + 1$. Neste caso, os coeficientes são 1011. Como o grau do polinômio é três, três zeros são adicionados aos dados para a divisão e o CRC será os três bits do resto da divisão, como ilustrado na Figura 3.

Segundo Fletcher (1982), mesmo que o CRC tenha propriedades de detecção de erros muito interessantes e podem ser implementados em hardware de modo muito eficiente, sua implementação por software não é eficiente. Uma implementação via software sem nenhuma otimização necessitaria fazer o uso de vários laços de repetição contendo operações de shift, o que não é eficiente em relação ao custo computacional, ou necessitaria fazer uso de *look-up tables*, que não é eficiente em relação ao uso de memória. Neste trabalho o CRC é implementado em software utilizando *look-up tables*. Mesmo que o CRC seja implementado desta forma, analisar seu custo computacional faz-se relevante visto que caso seja implementado em uma camada mais alta como a de aplicação, não há outra alternativa a não ser implementar o algoritmo em software. Além disso, atualmente a quantidade de memória principal em computadores atuais é abundante, fazendo com que seja preferida a implementação do algoritmo de CRC utilizando *look-up tables*.

Figura 3 – Exemplo de Cálculo do CRC com polinômio 1011 (grau 3)



Fonte: Negrizoli, Rodrigues e Oyamada (2021, p. 5)

Em geral, os códigos CRC de n bits detectam qualquer erro de um único bit, erros duplos, erros ímpares e rajadas de até n bits (AZAHARI et al., 2014), podendo detectar rajadas maiores com uma probabilidade de $1 - 2^{-n}$ (STONE et al., 1998). Como exemplo, o CRC-16 é um padrão para redes móveis (CDMA 2000) e LoRa (LongRange). Já o CRC-32 é usualmente utilizado nas transmissões da camada de enlace do padrão IEEE 802.3 (Ethernet) e 802.11 (Wi-Fi) (RAHMANI et al., 2007). No caso do padrão 802.3, o cálculo considera os dados de cabeçalho e dados. Cada protocolo pode utilizar um polinômio gerador específico, adequado de acordo com o meio de transmissão.

2.1.3 Checksum de Fletcher

O *Checksum* de Fletcher (FLETCHER, 1982) é especificado como sendo uma soma sucessiva de blocos do pacote de dados, semelhante ao *Checksum*. O tamanho do bloco de dados é definido pela metade do tamanho da soma (para o *Checksum* de Fletcher de 32 bits por exemplo, o tamanho da soma é de 16 bits). A soma é computada inicializando duas variáveis c_1 e c_2 com o valor zero e percorrendo bloco a bloco do pacote de dados. Para cada bloco o valor do bloco é acrescido em c_1 e após isso o valor de c_1 é acrescido em c_2.

Existem duas possíveis implementações para este algoritmo, a que utiliza aritmética de complemento de um e a que utiliza aritmética de complemento de dois para c_1 e c_2 (FLETCHER, 1982). Sendo k o tamanho de c_1 e c_2, na primeira implementação, considera-se que a cada soma o valor total é dividido em módulo por $2^k - 1$, e na segunda o valor total é dividido por 2^k. Caso um *overflow* ocorra, no primeiro caso é adicionado uma unidade ao bit menos significativo como consequência da operação de divisão em módulo por $2^k - 1$, no segundo caso o *overflow* é descartado. Conforme exposto pela Tabela 1, o *Checksum* de Fletcher produz duas somas. Essas somas, por fim, são concatenadas e enviadas ao destinatário do pacote de dados.

Tabela 1 – Exemplo de execução para o algoritmo de Fletcher com módulo $2^k - 1$ e blocos de 8 bits em que a soma é feita em complemento de um

Byte (B)	$c_1 = (c_{1_{anterior}} + B)\%255$	$c_2 = (c_{2_{anterior}} + c_1)\%255$
	0x00	0x00
0x12	0x12	0x12
0x10	0x22	0x34
0x0f	0x31	0x65
0x70	0xa1	0x07 (0x106 % 255)

Fonte: Produzido pelo autor

De acordo com Maxino (2006) a implementação que faz uso do complemento de um apresenta uma capacidade de detecção de erros superior à outra. Porém Nakassis (1988), diante dos resultados expostos na Tabela 2 indica que o método que utiliza complemento de dois falha em detectar uma porcentagem menor de erros, reforça que existem outras considerações que devem ser levadas em conta como o fato de que o tipo de erro que ocorre com mais frequência em um

dado canal de transmissão pode não ser detectado por uma ou outra implementação do algoritmo. Sendo assim, foram implementadas ao simulador ambas as versões do algoritmo de Fletcher para que seja realizado um comparativo de eficácia de detecção de erros independentemente do caráter dos erros produzidos pelo canal de transmissão considerado por Nakassis (1988). Além disto, faz-se relevante a comparação da eficácia e do custo computacional do *Checksum* de Fletcher com o algoritmo de CRC, visto que embora a capacidade de detecção de erros do primeiro possa se aproximar do CRC em alguns cenários mesmo sendo consideravelmente menos custoso computacionalmente.

Tabela 2 – Comparação das propriedades de detecção de erros de um CRC comum, o Checksum de Fletcher com mod(255) e mod(256) utilizando blocos de 8 bits

	CRC	Mod(255)	Mod(256)
Fração de erros não detectados	0,001526%	0,001538%	0,001526%
Fração de erros de rajadas de 16 bits não detectados	0,0%	0,000019%	0,0%
Erros de apenas um bit indetectados	0,0%	0,0%	0,0%
Distância mínima entre erros de dois bits não detectados	65535	2040	16

Fonte: Adaptado de Nakassis (1988)

2.1.4 Checksum de Adler

O *Checksum* de Adler (DEUTSCH; GAILLY, 1996) é idêntico à implementação do *Checksum* de Fletcher salvo o fato de que c_1 é inicializada com o valor um e o dividendo da operação de módulo realizada após cada soma não tem valor 2^k ou $2^k - 1$, mas sim o maior número primo possível que seja menor que $2^k - 1$ e de que este divide a sequência de dados em blocos de 8 bits, mesmo que os acumuladores c_1 e c_2 tenham 16 bits de tamanho. Em tese, o *Checksum* de Adler só está definido para uma chave de verificação de 32 bits, mas utilizando o conceito base de que o dividendo é o maior número primo até 2^k, pode-se definir o algoritmo para qualquer outro tamanho de chave de verificação. Isto é feito com a intenção de promover uma melhor distribuição dos bits da chave de verificação. Para a implementação do algoritmo de Adler com 32 bits de tamanho de chave de verificação usa-se o valor 65.521 como divisor, já que c_1 e c_2 possuem 16 bits de comprimento. Ambos são concatenados e enviados ao receptor assim como é feito no *Checksum* de Fletcher (MAXINO; KOOPMAN, 2009).

Segundo Mirza (2016), mesmo que uma melhor distribuição seja obtida ao utilizar este algoritmo, deve-se levar em conta que os padrões de erros capturados pelos valores entre 2^k e o maior número primo menor que 2^k são ignorados. Em termos gerais, é afirmado que o *Checksum* de Adler tem desempenho inferior ao de Fletcher e seu uso se mostra benéfico apenas em casos

onde erros de caráter específico ocorrem. Sendo assim o *Checksum* de Adler não é visto como uma troca benéfica em detrimento do *Checksum* de Fletcher ou o CRC (MIRZA, 2016). Embora o fato de que o *Checksum* de Adler é menos eficaz que o de Fletcher seja enfatizado nas literaturas estudadas, é relevante que esta afirmação seja posta à prova em diferentes cenários, já que não são dispostas informações sobre o padrão de erros injetados nos pacotes nos experimentos realizados pelos trabalhos citados.

2.2 Injeção de erros

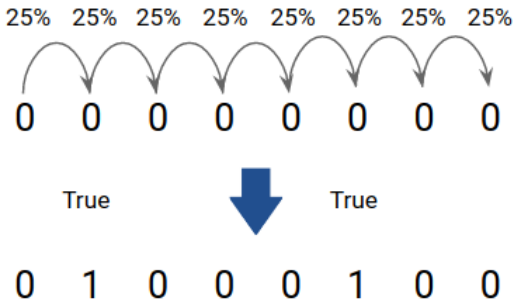
Para que seja realizado um estudo sobre a eficácia dos modelos de detecção de erros de forma empírica, devem ser usados modelos de erros de transmissão. Estes fazem com que erros de transmissão sejam produzidos de acordo com parâmetros de entrada que representam a frequência com que os erros de transmissão ocorrem de acordo com o algoritmo que define o comportamento do modelo de injeção de erros. Estes modelos são comuns na modelagem de interferências em meios de comunicação (TOURNOUX et al., 2011), sendo utilizados neste trabalho para a injeção de erros nos pacotes em nível de bit. Os modelos utilizados para estudo da eficácia dos algoritmos de detecção de erros são descritos a seguir de acordo com Negrizoli, Rodrigues e Oyamada (2021).

2.2.1 Modelo de Bernoulli

O Modelo de Bernoulli (JIANG; SCHULZRINNE, 2000) processa bit a bit o pacote de dados e para cada bit percorrido testa uma probabilidade previamente passada como parâmetro de entrada por meio de um gerador de números pseudo-aleatórios (BER). Caso o resultado do teste seja verdadeiro o valor do bit em questão é invertido e caso contrário o valor permanece o mesmo. Sendo assim, a quantidade de bits invertidos ao final do cômputo se aproxima do produto do tamanho do pacote em bits e a probabilidade do erro ocorrer. A principal característica deste modelo é a que a probabilidade de qualquer erro ocorrer em uma dada posição de bit do pacote não é dependente da probabilidade de ocorrer erro em nenhuma outra posição do pacote.

A Figura 4 exemplifica o processamento dos dados feito pelo Modelo de Bernoulli. Nela, podemos observar que o algoritmo percorre a mensagem bit a bit, e para cada bit presente na mensagem, é testada a probabilidade do bit ser negado (BER). Caso a probabilidade se faça verdadeira, o que ocorre duas vezes, o bit é negado simulando um erro de transmissão no bit em questão.

Figura 4 – Possibilidade de saída do Modelo de Bernoulli para $0x00$ como dados de entrada e 25% como taxa de erros

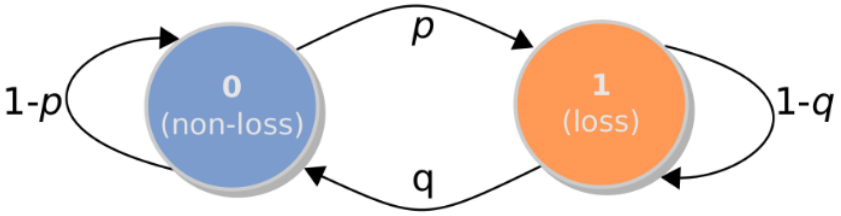


Fonte: Produzido pelo autor

2.2.2 Modelo de Gilbert

O Modelo de Gilbert, analogamente ao Modelo de Bernoulli, percorre o pacote de dados bit a bit e para cada bit uma probabilidade é testada. O que diferencia este modelo do anterior é que seu funcionamento ocorre de acordo com o estado em que seu autômato se encontra. Seu autômato é constituído de dois estados, *loss* e *non-loss*, quando seu autômato se encontra no estado *loss* o valor do bit em questão é alterado, caso esteja no estado *non-loss* o bit não é alterado. Após esta operação, o modelo de erros avança para o próximo bit no pacote de dados. O que define a passagem de estado são duas variáveis de entrada p e q. A variável p representa a probabilidade de passagem do estado *non-loss* para o estado *loss*, q analogamente representa a probabilidade de passagem do estado *loss* para o estado *non-loss* conforme ilustrado na Figura 5.

Figura 5 – Autômato que define o comportamento do Modelo de Gilbert



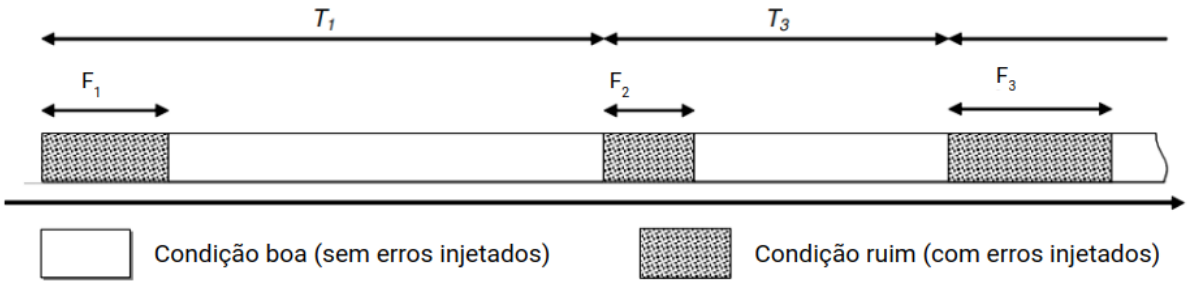
Fonte: Negrizoli, Rodrigues e Oyamada (2021, p. 6)

A frequência com que um dado tamanho de rajada ocorre nos dados de entrada está diretamente relacionada aos parâmetros de entrada q, já que este representa a probabilidade do autômato ir do estado de *loss* para o estado de *non-loss*. Sendo assim erros em rajada podem ser modelados escolhendo apropriadamente valores de p e q. Para produzir um cenário em que grande parte das rajadas de erros produzidas pelo Modelo de Gilbert são de tamanho K com uma taxa de erros e , usa-se $q = 1/K$ e $p = (q * e)/(1 - e)$.

2.2.3 Modelo Periódico

Outro modelo implementado e utilizado no simulador de envio de pacotes é o Modelo Periódico. Este modelo, diferentemente dos anteriores, possui uma abordagem determinística para decidir as posições em que os erros de transmissão ocorrem e qual o tamanho da rajada de erros. Neste modelo de erros, F (F_{Min} e F_{Max} para os casos onde o comprimento da rajada varia) denota o tamanho das rajadas que ocorrerão nos dados de entrada e T (T_{Min} e T_{Max} para os casos onde o comprimento do intervalo varia) denota a quantidade de bits sem nenhuma alteração que haverá entre uma rajada e outra (SOUSA; FERREIRA, 2007). O funcionamento do Modelo periódico se encontra ilustrado na Figura 6.

Figura 6 – Comportamento do Modelo Periódico



Fonte: Sousa e Ferreira (2007, p. 3)

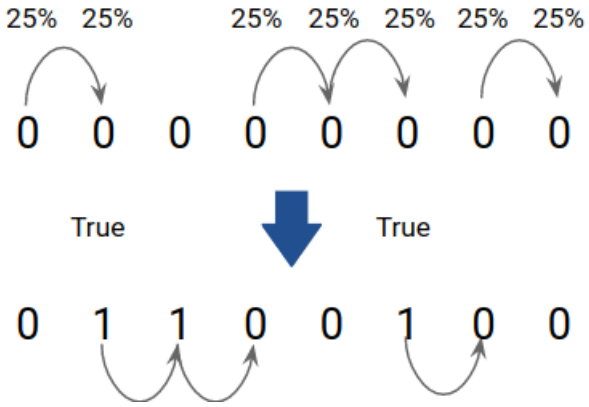
2.2.4 Modelo de Rajadas Esparsas

No trabalho desenvolvido por Negrizoli, Rodrigues e Oyamada (2021), foi avaliada a eficácia do polinômio $0x04C11DB7$ mediante experimentos que faziam uso dos modelos de Bernoulli, Gilbert e Periódico. Em um destes experimentos, notou-se que sua capacidade de detecção de erros era muito inferior ao esperado visto que é o polinômio utilizado na verificação de erros do protocolo Ethernet (SHEINWALD et al., 2002). Além disso, o polinômio proposto por Koopman (2015) não obteve um bom desempenho nos cenários em que o Modelo Periódico gerou os erros injetados nos pacotes de dados. Mediante a este cenário, foi suposto pelos autores do trabalho citado que o motivo deste comportamento pode ter sido consequência do caráter artificial dos erros produzidos pelos modelos de erro.

Sendo assim, para melhor investigar esse comportamento, é proposto neste trabalho um modelo de erros adicional que combine a característica do Modelo de Bernoulli, de que a ocorrência de um erro não é influenciada por outra, e a característica do Modelo Periódico, de produzir rajadas com com um tamanho bem definido, porém sem tornar determinístico o intervalo T_{min} e T_{max} em que cada rajada ocorre. Este modelo de erro propõe que cada posição de bit seja percorrida analogamente ao Modelo de Bernoulli testando uma probabilidade dada como entrada ao algoritmo que representa a taxa de ocorrência de rajadas (BOR) e caso o teste

seja verdadeiro, o algoritmo produz uma rajada de tamanho dentro do intervalo $[F_{min},F_{max}]$. Isto é observado na Figura 7, onde a cada bit é testada a probabilidade BOR, o que se faz verdadeira duas vezes, e em cada uma das duas ocorrências de rajada é produzida uma rajada de tamanho aleatório entre T_{min} e T_{max}. Deste modo é possível classificar este modelo como um modelo que produz rajadas de forma esparsa.

Figura 7 – Possibilidade de saída do Modelo de Rajadas Esparsas para $0x00$ como dados de entrada, $BOR = 0.25$ e $F = [1, 2]$



Fonte: Produzido pelo autor

Como consequência deste comportamento, nota-se que a quantidade média de erros produzidos é de $N * BOR * F$ sendo N o tamanho do pacote e F a mediana do intervalo nos casos em que F não é de tamanho fixo considerando que a probabilidade de todos os números dentro do intervalo F serem sorteados é a mesma.

3

Materiais e Métodos

Para avaliar a eficácia dos algoritmos apresentados no Capítulo 2, isto é, o *Checksum*, o CRC, o *Checksum* de Fletcher com aritmética de complemento de um e complemento de dois, e o *Checksum* de Adler, será utilizado o simulador de transmissão e injeção de erros em pacotes PacketSenderSim (NEGRIZOLI; RODRIGUES; OYAMADA, 2021). O simulador é desenvolvido em C++ e permite adquirir dados estatísticos referentes à eficácia dos algoritmos verificação criando pacotes com diferentes tamanhos, injetando erros de transmissão de acordo com os modelos de injeção de erros e pondo à prova a capacidade de detecção de erros destes algoritmos [1].

3.1 Mudanças aplicadas ao simulador

Embora o simulador tenha sido previamente desenvolvido, foram necessárias a realização de mudanças significativas em seu código. A rotina de teste de eficácia foi completamente refatorada com o objetivo de deixar o código mais enxuto, diminuir seu tempo de execução e permitir que múltiplas combinações de algoritmos de detecção e de injeção de erros sejam testados em apenas uma execução, sem necessitar executar o programa simulador múltiplas vezes com parâmetros diferentes. Também foi adicionado ao simulador o modelo de rajadas esparsas, a rotina responsável por medir o tempo de execução dos algoritmos de detecção de erros e uma rotina de testes alternativa que compara a eficácia de dois algoritmos, disponibilizando estatísticas relacionadas não apenas ao desempenho de cada um dos algoritmos individualmente, mas também dados referentes a um cenário em que os dois sejam utilizados em conjunto. Mais detalhes à respeito do funcionamento dessas rotinas de teste estão dispostas na Subseção 3.2.2.

Além disso o simulador foi desenvolvido com enfoque na programação orientada À objetos, tornando-o altamente extensível. Isto permite que contribuições no código por terceiros

[1] Código-fonte disponível em <https://github.com/igorFNegrizoli/packetSenderSim>

seja feita com grande facilidade. Para implementar um novo modelo de erros e passá-lo como parâmetro para as rotinas de teste basta derivar uma classe filha do *template* utilizado para definir os comportamentos padrões dos modelos de erros. Desta forma o contribuinte não precisa se preocupar com detalhes específicos do funcionamento do simulador, dedicando-se apenas ao desenvolvimento do novo modelo de erros.

Além disso, foram foram implementados os algoritmos de detecção de erros citados anteriormente que não estavam presentes no simulador (*Checksum* de Fletcher e de Adler) e o algoritmo de *Checksum* simples foi refatorado com o objetivo de melhorar o tempo de execução. O algoritmo de CRC, por sua vez, sofreu mudanças significativas já que apresentou inconsistências nos resultados apresentados por Negrizoli, Rodrigues e Oyamada (2021). A refatoração do CRC foi baseada no código disponibilizado na plataforma GitHub publicamente por Stancliff (2020) (para chave de 16 bits) e Pohoreski (2017) (para chave de 32 bits), ambas implementações fazem uso de *Lookup Tables* para diminuir o tempo de cômputo das chaves de verificação. Em ambos repositórios constam vários métodos distintos que geram os códigos de verificação desejados, a implementação utilizada foi a CRC-16 (CCITT) para o CRC de 16 bits e ITU I.363.5 para o CRC de 32 bits. É importante salientar que a escolha do método não possui impacto na capacidade de detecção de erros, apenas na chave de verificação gerada pelo algoritmo, portanto quaisquer outros métodos poderiam ter sido escolhidos sem impactar nos resultados deste trabalho.

Este simulador é implementado seguindo o paradigma de Orientação a Objetos, sendo assim, suas funcionalidades podem ser muito bem representadas pelo diagrama de classes da aplicação que se faz presente no Apêndice B junto a uma breve descrição de seus funcionalidades. O diagrama classes que representa o simulador antes das mudanças propostas por este trabalho serem aplicadas se faz presente no Apêndice A.

3.2 Funcionamento do simulador

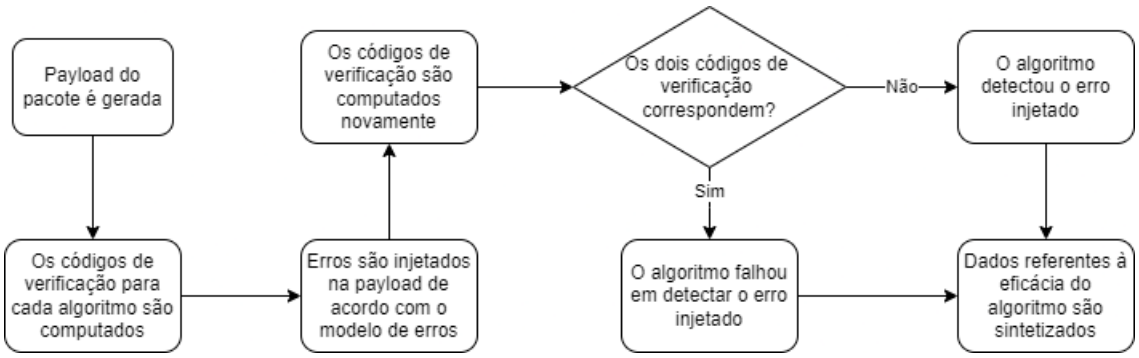
Nesta seção será discutido o funcionamento do simulador, como fazer uso de suas rotinas de teste, e como interpretar as saídas dispostas por ele. Um código exemplo acompanhado de explicações a respeito do uso do *framework* se faz presente no Apêndice C. Para a compilação do projeto é utilizado o nível de otimização $-O2$. A utilização do nível de otimização $-O3$ foi descartada após testes preliminares serem realizados e ter sido constatado que em alguns casos o ganho foi mínimo e em vários casos não houve ganho de desempenho dos algoritmos de verificação.

É importante apontar que todos os experimentos realizados no simulador são completamente reprodutíveis desde que a semente utilizada no gerador de números aleatórios seja a mesma.

3.2.1 Workflow

Para simular um erro ocorrido na transmissão de pacotes, as rotinas de teste implementadas primeiramente geram o conteúdo do pacote e após isso geram os códigos de verificação para cada algoritmo analisado. Após isso, são injetados erros no pacote de acordo com o algoritmo de injeção de erros desejado. Como é certo que erros ocorreram após o primeiro cômputo da chave de verificação, ao calcular a chave de verificação pela segunda vez espera-se que o algoritmo produza uma chave diferente da anterior. Caso a chave produzida pelo algoritmo de verificação após a injeção de erros seja idêntica à chave produzida antes da injeção de erros significa que o algoritmo falhou em detectar as mudanças ocorridas no pacote. É possível observar esse fluxo de trabalho representado em forma de fluxograma na Figura 8. Este processo é repetido várias vezes, com isso é possível obter uma contagem dos cenários em que o algoritmo de verificação falha em determinado cenário definido pelo modelo de injeção de erros.

Figura 8 – Fluxograma ilustrando o *workflow* da simulação de erros



Fonte: Produzido pelo autor

3.2.2 Funcionamento das rotinas

O simulador possui três principais rotinas de teste: *genericTest*, *executionTimeTest* e *compareTwoAlgorithms*. O funcionamento dessas rotinas serão apresentados a seguir. É importante salientar que as saídas dispostas pelas rotinas de teste realizam os testes para tamanhos de pacote variando desde 8 Bytes até 1024 Bytes, variando em potência de dois.

a) *genericTest*

Esta rotina de teste possui cinco entradas: um vetor contendo os algoritmos de verificação que terão suas eficácias testadas, os modelos de erros que serão aplicados para testar o conjunto de algoritmos de verificação, o tamanho do vetor que armazena os algoritmos de verificação, o tamanho do vetor que armazena os modelos de erros e um gerador de números aleatórios (para geração de pacotes). A *genericTest*, para cada modelo de erros presente no conjunto de entrada e para cada algoritmo de verificação,

o processo definido na Figura 8 é repetido n vezes, sendo que n é um parâmetro de entrada para o construtor da classe *TestRoutines*. Após n iterações ocorridas a rotina imprime no terminal do usuário uma tabela com as seguintes informações para cada tamanho de pacote ($N(B)$): média de bits invertidos por pacote ($f(bit)$), porcentagem de bits invertidos por pacote (%), e quantas das n iterações não apresentaram falhas de detecção para cada algoritmo presente no parâmetro de entrada. Um exemplo de saída é apresentado na Figura 9, uma tabela deste modelo é produzida para cada modelo de erros presente no parâmetro de entrada.

Figura 9 – Saída disposta pela rotina *genericTest*

```
Sparse Bursts Error Model:
BOR: 0.00100
F: [2, 8]

N(B)    f(bit)  %       CHK16   CHK32   CRC16   CRC16   CRC32   CRC32   CRC32   1CFL16  2CFL16  ADLR16  1CFL32  2CFL32  ADLR32
8       1.24    1.94    3       0       0       0       0       0       0       15      0       8       0       0       0
16      1.51    1.18    15      6       0       0       0       0       0       12      1       15      0       0       0
32      2.04    0.80    45      26      0       4       0       0       0       35      2       26      0       1       0
64      3.14    0.61    146     64      0       1       0       0       0       32      5       45      0       0       0
128     5.44    0.53    344     170     0       12      0       0       0       65      26      34      0       5       0
256     10.30   0.50    577     250     11      12      0       0       0       52      47      39      0       9       0
512     20.38   0.50    392     166     22      21      0       0       0       32      31      26      0       1       0
1024    40.76   0.50    49      11      13      11      0       0       0       19      18      17      0       0       0
```

Fonte: Produzido pelo autor

b) *executionTimeTest*

A rotina *executionTimeTest* possui três entradas: um vetor contendo os algoritmos de verificação que terão seus tempos de execução medidos, o tamanho deste vetor, e um gerador de números aleatórios (para geração de pacotes). Para cada algoritmo presente no vetor de entrada, é gerada uma tabela com a medição de tempo em milissegundos para o cômputo dos códigos de verificação de n pacotes, sendo n especificado nos parâmetros de entrada do construtor da classe *TestRoutines*. Esta estatística é disposta para cada tamanho de pacote conforme disposto na Figura 10.

Visto que o tempo de execução de todos os algoritmos implementados varia conforme o tamanho do pacote e não depende de modo algum do conteúdo do pacote, é utilizado o mesmo pacote para todas as n iterações, fazendo assim com que a execução dos testes seja mais rápida já que não é preciso produzir um pacote diferente n vezes para cada tamanho de pacote.

c) *compareTwoAlgorithms*

Esta rotina rotina possui cinco entradas: um vetor contendo os modelos de erros que serão aplicados para testar os dois algoritmos de verificação, o tamanho deste vetor, os dois algoritmos a serem comparados, e um gerador de números aleatórios (para a geração de pacotes). A rotina produz n pacotes, sendo n especificado na instanciação da classe *TestRoutines*, e para cada pacote, injeta erros de transmissão de acordo com o cada modelo de erros especificado. Para cada modelo de erros é gerada uma

Figura 10 – Saída disposta pela rotina *executionTimeTest*

```
CHK16
N(B)    Time(ms)
8       12
16      18
32      29
64      55
128     96
256     179
512     346
1024    678
```

Fonte: Produzido pelo autor

tabela, que para cada tamanho de pacote exibe no terminal do usuário a média de bits invertidos por pacote, a porcentagem de bits invertidos por pacote, a quantidade de falhas de detecção de cada um dos dois algoritmos comparados, e a quantidade de falhas de detecção que ocorreram em ambos os algoritmos. Um exemplo de saída está disposto na Figura 11.

Figura 11 – Saída disposta pela rotina *compareTwoAlgorithms*

```
Bernoulli Error Model:
BER: 0.00010

N(B)    f(bit)  %       CHK32   2CFL32  BOTH
8       1.00    1.56    0       0       0
16      1.00    0.78    1       1       1
32      1.00    0.39    3       1       1
64      1.00    0.20    14      3       1
128     1.01    0.10    88      12      6
256     1.02    0.05    274     52      30
512     1.07    0.03    870     160     123
1024    1.26    0.02    2359    415     283
```

Fonte: Produzido pelo autor

3.3 Ambiente de testes

Durante a execução dos testes, processos que não dizem respeito à execução do simulador foram encerrados com o objetivo de obter uma maior consistência na obtenção do custo computacional dos algoritmos. A máquina em que foram executados os testes será um computador pertencente ao Programa de Educação Tutorial de Ciência da Computação da Universidade

Estadual do Oeste do Paraná (PETComp), as especificações relevantes para o experimento são:

- Processador: Intel I3-10100F 3,6 GHz
- Memória principal: 16GB (2.666Mhz)
- Sistema operacional: Ubuntu 20.04 LTS
- Shell: bash
- Compilador utilizado: g++

3.4 Cenários de teste

Visto que o principal objetivo deste trabalho é realizar uma análise da eficácia dos algoritmos de detecção de erros, todos os algoritmos apresentados na Seção 2.1 terão sua capacidade de detecção de erros testada, sendo eles apresentados na Tabela 3. Para que a eficácia destes algoritmos seja devidamente testada, serão executados 21 casos de teste com base nos modelos de injeção de erros de transmissão apresentados na Seção 2.2, sendo eles apresentados na Tabela 4. Essa coleta de dados será feita através do uso da rotina de testes *genericTest*.

Dentre os polinômios utilizados na Tabela 3, o polinômio $0x1021$ é utilizado como polinômio padrão pelo CCITT, o polinômio $0x04c11db7$ é utilizado pelo protocolo Ethernet, e os demais são propostos por Koopman (2015) como alguns dos melhores polinômios possíveis para o cômputo de códigos de CRC.

Tabela 3 – Algoritmos de detecção de erros utilizados e seus parâmetros de entrada

	Algoritmo	**Parâmetro**
1	Checksum16Bit	-
2	Checksum32Bit	-
3	CRC16Bit	G(x) = 0x1021
4	CRC16Bit	G(x) = 0x8d95
5	CRC32Bit	G(x) = 0x04c11db7
6	CRC32Bit	G(x) = 0xc9d204f5
7	CRC32Bit	G(x) = 0x973afb51
8	Fletcher16Bit	Complemento de um
9	Fletcher16Bit	Complemento de dois
10	Adler16Bit	-
11	Fletcher32Bit	Complemento de um
12	Fletcher32Bit	Complemento de dois
13	Adler32Bit	-

Fonte: Produzido pelo autor

Além disso serão coletados dados a respeito do tempo de execução de todos os algoritmos apresentados na Tabela 3 utilizando a rotina *executionTimeTest*. Quanto à comparação dois a dois

Tabela 4 – Modelos de injeção de erros utilizados na rotina *genericTest* e seus parâmetros de entrada

	Algoritmo	**Parâmetro**
1	SparseBurstsErrorModel	BOR = 0,001 | F = [2, 8]
2	SparseBurstsErrorModel	BOR = 0,0001 | F = [16, 32]
3	SparseBurstsErrorModel	BOR = 0,001 | F = [32, 64]
4	SparseBurstsErrorModel	BOR = 0,0001 | F = [32, 64]
5	SparseBurstsErrorModel	BOR = 0,001 | F = [64, 128]
6	SparseBurstsErrorModel	BOR = 0,0001 | F = [64, 128]
7	BernoulliErrorModel	BER = 0,01
8	BernoulliErrorModel	BER = 0,001
9	BernoulliErrorModel	BER = 0,0001
10	GilbertErrorModel	K = 2 | e = 0,001
11	GilbertErrorModel	K = 2 | e = 0,0001
12	GilbertErrorModel	K = 8 | e = 0,001
13	GilbertErrorModel	K = 8 | e = 0,0001
14	GilbertErrorModel	K = 16 | e = 0,001
15	GilbertErrorModel	K = 16 | e = 0,0001
16	GilbertErrorModel	K = 32 | e = 0,001
17	GilbertErrorModel	K = 32 | e = 0,0001
18	PeriodicBurstErrorModel	F = [1, 1] | T = [16, 16]
19	PeriodicBurstErrorModel	F = [1, 3] | T = [16, 16]
20	PeriodicBurstErrorModel	F = [8, 16] | T = [32, 64]
21	PeriodicBurstErrorModel	F = [16, 32] | T = [128, 256]

Fonte: Produzido pelo autor

dos algoritmos, será comparado o uso do *Checksum* em conjunto com os seguintes algoritmos: CRC, Fletcher com complemento de um, Fletcher com complemento de dois, e Adler. Estas combinações são feitas tanto para os algoritmos de verificação que produzem chaves de 16 bits quanto para os que produzem chaves de 32 bits. Serão utilizados 4 modelos de erros para cada comparação, estes modelos estão presentes na Tabela 5.

Tabela 5 – Modelos de injeção de erros utilizados na rotina *compareTwoAlgorithms* e seus parâmetros de entrada

	Modelo	Parâmetros
1	SparseBurstsErrorModel	BOR = 0,0001 | F = [32, 64]
2	BernoulliErrorModel	BER = 0,0001
4	GilbertErrorModel	K = 16 | e = 0,0001
5	PeriodicBurstErrorModel	F = [8, 16] | T = [32, 64]

Fonte: Produzido pelo autor

4

Resultados e Discussão

Após realizados os testes descritos no Capítulo 3, foi feita uma análise dos cenários em que os algoritmos falham em detectar erros de transmissão e suas vantagens e desvantagens. As discussões serão particionadas de acordo com os modelos de erros utilizados para que as discussões sobre a eficácia dos algoritmos sejam condensadas em cenários similares.

É importante salientar que dentre os vários testes realizados, apenas uma parte destes testes é disposta neste trabalho, isto se deve à grande quantidade de testes realizados. Todavia, estes testes são agentes influenciadores nas conclusões tidas acerca dos itens analisados neste capítulo, mesmo que apenas alguns exemplos sejam dispostos. Todos os testes realizados, incluindo os que não foram apresentados explicitamente neste trabalho estão dispostos em um documento de texto no repositório online[1].

O simulador não indica qual o polinômio utilizado pelo algoritmo de CRC nas colunas da tabela de saída. Para ler a tabela, deve-se considerar que os algoritmos estão na mesma ordem em que os algoritmos da Tabela 3. Portanto, os polinômios usados nas duas colunas que possuem CRC16 como título têm $0x1021$ e $0x8d95$ como polinômios geradores, nesta exata ordem. E de modo análogo, os polinômios utilizados nas três colunas que possuem CRC32 como título têm $0x04c11db7$, $0xc9d204f5$ e $0x973afb51$ como polinômios geradores, respectivamente.

4.1 Modelo de Rajadas Esparsas

Quanto aos cenários em que o Modelo de Rajadas Esparsas foi aplicado, percebe-se que os algoritmos de detecção de erros que operam utilizando somas sucessivas e com tamanho de chaves de verificação igual ou menor que o tamanho mínimo das rajadas produzidas tem desempenho mais satisfatório conforme a probabilidade de ocorrência de rajadas diminui. Este

[1] Disponível em <https://github.com/igorFNegrizoli/packetSenderSim/blob/master/testeOutput.txt>

comportamento pode ser notado nas colunas *CHK16*, *1CFL16*, *2CFL16* e *ADLR16* na Tabela 6 e na Tabela 7.

Tabela 6 – Eficácia dos algoritmos com erros injetados pelo modelo de rajadas esparsas

	BOR = 0.001 | F = [32, 64]														
N(B)	f(bit)	%	CHK16	CHK32	CRC16	CRC16	CRC32	CRC32	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	2.81	4.39	1	0	0	0	0	0	0	9	0	7	0	0	0
16	5.69	4.44	2	0	0	0	0	0	0	10	0	11	0	0	0
32	11.44	4.47	4	0	0	0	0	0	0	9	4	9	0	0	0
64	22.97	4.49	2	0	0	0	0	0	0	12	4	10	0	0	1
128	46.23	4.51	5	0	3	7	0	0	0	13	7	8	0	0	0
256	92.80	4.53	11	0	5	5	0	0	0	15	10	14	0	0	0
512	186.55	4.55	16	0	9	11	0	0	0	14	17	16	0	0	0
1024	373.96	4.57	14	0	15	15	0	0	0	17	21	19	0	0	0

Fonte: Produzido pelo autor

Tabela 7 – Eficácia dos algoritmos com erros injetados pelo modelo de rajadas esparsas com uma taxa de ocorrência de rajadas menor

	BOR = 0.0001 | F = [32, 64]														
N(B)	f(bit)	%	CHK16	CHK32	CRC16	CRC16	CRC32	CRC32	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	1.18	1.85	0	0	0	0	0	0	0	0	0	1	0	0	0
16	1.48	1.15	0	0	0	0	0	0	0	0	0	0	0	0	0
32	2.07	0.81	0	0	0	0	0	0	0	1	1	0	0	0	0
64	3.28	0.64	0	0	0	0	0	0	0	0	0	0	0	0	0
128	5.69	0.56	2	0	0	0	0	0	0	2	3	2	0	0	0
256	10.49	0.51	5	0	0	0	0	0	0	1	4	5	0	0	0
512	20.11	0.49	5	0	0	1	0	0	0	4	8	3	0	0	0
1024	39.40	0.48	10	0	6	6	0	0	0	7	9	7	0	0	0

Fonte: Produzido pelo autor

Também é possível perceber certa robustez dos algoritmos de detecção com palavras de verificação de 32 bits mesmo no cenário em que a taxa de ocorrência de rajadas é mais elevada. Até mesmo os algoritmos de detecção baseados em soma cujo o desempenho esperado não era muito alto como o *Checksum*, *Checksum de Fletcher* e *Checksum de Adler* se saíram tão bem quanto o algoritmo de CRC nestas situações apesar de exigirem um poder de processamento bem menor.

O CRC de 16 bits por outro lado, não apresentou desempenho satisfatório nas situações em que existem múltiplas rajadas no pacote, que é o caso de pacotes de maior tamanho dos cenários em que $BOR = 0.001$. O CRC de 32 bits, como é possível observar na Tabela 8, não apresenta tal comportamento mesmo quando o tamanho das rajadas é aumentado.

4.2 Modelo de Bernoulli

Com o modelo de Bernoulli, é possível identificar na Tabela 9, Tabela 10 e Tabela 11 uma das características que mais fazem os algoritmos de verificação baseados em somas sucessivas serem menos eficazes em detectar erros que os algoritmos baseados em divisão de polinômios: a dificuldade em detectar erros unitários alinhados na mesma posição na palavra. Como o Modelo de Bernoulli produz erros unitários, a chance de produzir erros como duas palavras distintas

Tabela 8 – Eficácia dos algoritmos com erros injetados pelo modelo de rajadas esparsas com um tamanho de rajadas maior que 32 bits

	BOR = 0.0001 | F = [64, 128]														
N(B)	f(bit)	%	CHK16	CHK32	CRC16	CRC16	CRC32	CRC32	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	1.20	1.88	0	0	0	0	0	0	0	2	0	1	0	0	0
16	1.73	1.35	0	0	0	0	0	0	0	0	1	0	0	0	0
32	2.94	1.15	0	0	0	0	0	0	0	1	0	0	0	0	0
64	5.37	1.05	1	0	0	0	0	0	0	1	0	0	0	0	0
128	10.18	0.99	3	0	0	0	0	0	0	1	1	3	0	0	0
256	19.89	0.97	2	0	1	1	0	0	0	2	3	2	0	0	0
512	39.14	0.97	6	0	1	0	0	0	0	9	10	12	0	0	0
1024	78.01	0.95	11	0	3	1	0	0	0	8	9	10	0	0	0

Fonte: Produzido pelo autor

sofrendo injeção de erro na mesma posição de bit, aumenta drasticamente a chance de falhas de detecção acontecerem ao utilizar os algoritmos baseados em somas sucessivas. É possível notar também que conforme a taxa de ocorrência de erros diminui, as ocorrências de erros aumentam nos pacotes de tamanho maior e diminuem nos de tamanho menor. Isso ocorre pois conforme a ocorrência de erros diminui em pacotes de tamanho pequeno, menor a probabilidade dos erros presentes no pacote se alinharem na mesma posição de bit. De modo análogo, pacotes maiores são mais suscetíveis à falha de detecção de erros dos algoritmos baseados em somas sucessivas, já que com menos bit invertidos, mais provável é que ocorra tal alinhamento.

Tabela 9 – Eficácia dos algoritmos com erros injetados pelo modelo de Bernoulli com $BER = 0.01$

	BER = 0.01														
N(B)	f(bit)	%	CHK16	CHK32	CRC16	CRC16	CRC32	CRC32	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	1.16	1.82	2672	897	0	0	0	0	0	13	807	13	4	115	13
16	1.56	1.22	6837	2860	2	17	0	0	0	38	2204	35	16	403	34
32	2.64	1.03	8590	3713	4	34	0	0	0	60	2945	57	28	619	52
64	5.13	1.00	3507	1351	9	18	0	0	0	40	1232	33	13	237	24
128	10.24	1.00	266	44	19	16	0	0	0	17	88	21	0	2	1
256	20.49	1.00	24	0	17	15	0	0	0	12	23	12	0	0	0
512	40.96	1.00	16	0	12	18	0	0	0	12	15	12	0	0	0
1024	81.92	1.00	15	0	14	14	0	0	0	22	9	11	0	0	0

Fonte: Produzido pelo autor

Tabela 10 – Eficácia dos algoritmos com erros injetados pelo modelo de Bernoulli com $BER = 0.001$

	BER = 0.001														
N(B)	f(bit)	%	CHK16	CHK32	CRC16	CRC16	CRC32	CRC32	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	1.00	1.57	42	13	0	0	0	0	0	0	9	0	0	0	0
16	1.01	0.79	218	97	0	0	0	0	0	0	55	0	0	5	0
32	1.03	0.40	745	331	0	0	0	0	0	0	237	0	0	43	0
64	1.11	0.22	2468	1152	0	2	0	0	0	2	848	2	1	196	2
128	1.38	0.13	6082	2986	1	2	0	0	0	9	2228	5	3	493	5
256	2.18	0.11	9283	4339	6	5	0	0	0	16	3357	15	0	761	2
512	4.11	0.10	5635	2487	9	21	0	0	0	51	2019	53	1	405	5
1024	8.19	0.10	797	211	12	16	0	0	0	22	258	22	0	39	0

Fonte: Produzido pelo autor

Além disso, é possível observar a alta capacidade dos algoritmos de CRC em detectar erros únicos, tendo sua performance prejudicada conforme a quantidade de bits invertidos por

Tabela 11 – Eficácia dos algoritmos com erros injetados pelo modelo de Bernoulli com $BER = 0.0001$

	BER = 0.0001														
N(B)	f(bit)	%	CHK16	CHK32	CRC16	CRC16	CRC32	CRC32	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	1.00	1.56	0	0	0	0	0	0	0	0	0	0	0	0	0
16	1.00	0.78	4	2	0	0	0	0	0	0	1	0	0	0	0
32	1.00	0.39	7	1	0	0	0	0	0	0	2	0	0	1	0
64	1.00	0.20	37	16	0	0	0	0	0	0	11	0	0	2	0
128	1.01	0.10	144	80	0	0	0	0	0	0	53	0	0	16	0
256	1.02	0.05	541	269	0	0	0	0	0	0	188	1	0	52	0
512	1.07	0.03	1694	836	0	0	0	0	0	12	650	5	0	141	0
1024	1.26	0.02	4923	2424	0	0	0	0	0	21	1838	23	0	450	0

Fonte: Produzido pelo autor

pacote aumenta. Este comportamento pode ser observado com mais clareza na Tabela 11, na qual para todos os tamanho de pacote, a média de bits invertidos se aproxima de uma unidade e não ocorre nenhuma falha de detecção em nenhum dos algoritmos de CRC. Também é possível observar que as implementações de 32 bits do *Checksum* de Fletcher com complemento de um e do *Checksum* de Adler apresentam desempenho similar.

4.3 Modelo de Gilbert

Diante dos resultados do modelo de Gilbert, é possível observar comportamentos identificados já na Seção 4.1, como a ineficiência dos algoritmos baseados em somas sucessivas em detectar rajadas frequentes devido à alta probabilidade de alinhamento dessas rajadas. Este comportamento pode ser observado na Tabela 12, onde conforme a taxa de bits invertidos por pacote se torna maior que o parâmetro K (tamanho médio da rajada), múltiplas rajadas de tamanho K são produzidas, tendo elas alta probabilidade de alinhamento.

Tabela 12 – Eficácia dos algoritmos com erros injetados pelo modelo de Gilbert com $e = 0.001$ e $K = 2$

	e = 0.001 | K = 2														
N(B)	f(bit)	%	CHK16	CHK32	CRC16	CRC16	CRC32	CRC32	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	1.03	1.61	7	0	0	0	0	0	0	0	2	1	0	0	0
16	1.06	0.83	27	14	0	0	0	0	0	0	9	0	0	1	0
32	1.14	0.44	88	42	0	0	0	0	0	2	16	1	0	5	0
64	1.28	0.25	335	156	0	0	0	0	0	2	103	1	1	25	0
128	1.62	0.16	1238	606	0	1	0	0	0	4	318	2	0	74	0
256	2.41	0.12	3048	1466	0	0	0	0	0	7	739	6	0	165	1
512	4.22	0.10	4541	2027	8	7	0	0	0	29	1147	30	3	248	3
1024	8.20	0.10	2808	1201	12	12	0	0	0	36	655	38	1	133	2

Fonte: Produzido pelo autor

Diante dos resultados obtidos também foi possível notar a incapacidade do modelo de erros de gerar alguns cenários propostos. Nos cenários em que se deseja gerar grandes rajadas com baixa ocorrência, o algoritmo falha em gerar tais rajadas devido ao fato de que a probabilidade da rajada ser interrompida quando o algoritmo está em estado de *loss* cresce conforme o parâmetro e decresce. Este problema pode ser observado na Tabela 13, onde a média de bits invertidos por

pacote ($f(bit)$) se mantém bem abaixo do valor de K mesmo em pacotes maiores, indicando assim que o tamanho das rajadas produzidas não se aproximam nem um pouco do valor de K especificado. Isso demonstra o quão importante é o desenvolvimento do Modelo de Rajadas Esparsas, para produzir que cenários em que o tamanho da rajada seja bem definido e ainda sim sua taxa de ocorrência possa ser baixa sem prejudicar a fidelidade dos erros produzidos aos parâmetros de entrada.

Tabela 13 – Eficácia dos algoritmos com erros injetados pelo modelo de Gilbert com $e = 0.001$ e $K = 32$

			e = 0.001 | K = 32												
N(B)	f(bit)	%	CHK16	CHK32	CRC16	CRC16	CRC32	CRC32	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	1.03	1.62	0	0	0	0	0	0	0	0	0	0	0	0	0
16	1.10	0.86	1	0	0	0	0	0	0	1	0	0	1	0	0
32	1.22	0.48	0	0	0	0	0	0	0	1	0	1	0	0	0
64	1.47	0.29	0	0	0	0	0	0	0	2	0	0	0	0	0
128	1.98	0.19	1	0	0	0	0	0	0	1	0	2	0	0	0
256	2.96	0.14	1	1	0	0	0	0	0	1	0	2	0	0	0
512	4.95	0.12	1	1	0	0	0	0	0	3	4	6	0	0	0
1024	9.02	0.11	8	3	2	0	0	0	0	4	2	11	0	0	0

Fonte: Produzido pelo autor

4.4 Modelo de Rajadas Periódicas

Conforme dito na Subseção 2.2.3, o Modelo de Rajadas Periódicas possui uma abordagem bem determinística para produzir erros injetados. Ao definir os cenários de teste que dizem respeito à este modelo de erros foram definidos tanto cenários que visam mitigar esta característica quanto cenários em que esta característica é explorada com o intuito de evidenciar falhas nos algoritmos de detecção.

Um cenário em que pode-se enxergar com maior clareza uma das fraquezas dos algoritmos baseados em soma é o exposto na Tabela 14. Nela, pode-se observar o quão prejudicial erros alinhados podem ser para estes algoritmos, que se mostram nem um pouco adequados para cenários em que rajadas de erros ocorrem em um intervalo bem definido. É importante observar que as falhas de detecção diminuem conforme o tamanho dos pacotes aumenta, já que a probabilidade de alinhamento de todos os bits invertidos diminui. Ainda sim, observa-se que os algoritmos de Fletcher em complemento de um e Adler se saem bem melhor que o esperado, suas versões de 16 bits surpreendentemente se saem melhor que a versão de 32 bits do *Checksum* convencional. Isso evidencia que mesmo os algoritmos baseados em soma podem empregar mecanismos que mitigam falhas inerentes ao modo com que suas palavras chave são calculadas.

Diante dos testes realizados também foi possível identificar na Tabela 15 que mesmo em cenários em que as rajadas são mais distribuídas através dos pacotes produzidos os algoritmos baseados em soma que produzem palavras de 16 bits ainda sofrem falhas de detecção. Não obstante, é possível observar que os algoritmos de *Checksum*, Fletcher e Adler que produzem

Tabela 14 – Eficácia dos algoritmos com erros injetados pelo modelo de Rajadas Periódicas com $T = [16, 16]$ e $F = [1, 3]$

	T = [16, 16] | F = [1, 3]														
N(B)	f(bit)	%	CHK16	CHK32	CRC16	CRC16	CRC32	CRC32	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	8.00	12.50	47783	9560	0	0	0	0	0	2412	44711	2412	2412	22374	2412
16	16.00	12.50	31494	2259	0	0	0	0	0	436	32293	436	436	15969	436
32	32.00	12.50	22612	961	18	11	0	0	0	82	31594	85	82	15901	82
64	64.00	12.50	16253	499	11	20	0	0	0	70	31488	68	24	15741	24
128	128.00	12.50	11495	275	18	10	0	0	0	48	31512	50	4	15837	4
256	256.01	12.50	8065	141	19	14	0	0	0	38	31250	28	1	15600	1
512	511.98	12.50	5622	67	15	12	0	0	0	26	31098	22	0	15589	0
1024	1024.02	12.50	4065	35	15	16	0	0	0	14	31287	25	0	15727	0

Fonte: Produzido pelo autor

chaves de verificação de 32 bits apresentam um desempenho satisfatório mesmo neste cenário em que suas variantes de 16 bits apresentam falhas de detecção.

Tabela 15 – Eficácia dos algoritmos com erros injetados pelo modelo de Rajadas Periódicas com $T = [128, 256]$ e $F = [16, 32]$

	T = [128, 256] | F = [16, 32]														
N(B)	f(bit)	%	CHK16	CHK32	CRC16	CRC16	CRC32	CRC32	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	24.00	37.49	10	0	0	0	0	0	0	14	3	18	2	0	0
16	23.99	18.75	15	0	0	0	0	0	0	12	4	18	2	0	0
32	45.59	17.81	12	0	0	0	0	0	0	15	19	19	0	0	0
64	74.85	14.62	17	0	15	13	0	0	0	19	14	16	0	0	0
128	138.94	13.57	12	0	17	15	0	0	0	15	19	12	0	0	0
256	266.93	13.03	17	0	12	19	0	0	0	11	8	12	0	0	0
512	522.91	12.77	14	0	7	15	0	0	0	8	20	17	0	0	0
1024	1034.85	12.63	16	0	19	16	0	0	0	17	8	23	0	0	0

Fonte: Produzido pelo autor

4.5 Custo computacional

Após realizados os testes de custo computacional dos algoritmos de verificação os seguintes resultados foram obtidos. Visto que a implementação utilizada para os algoritmos de CRC emprega a técnica de *lookup tables*, o algoritmo não tem sua complexidade dependente do conjunto de dados nem de $G(x)$ (polinômio gerador), logo não se faz necessário a verificação do desempenho desses algoritmos para diferentes polinômios. Os polinômios utilizados são $0x1021$ para o CRC de 16 bits e $0x04C11DB7$ para o CRC de 32 bits. A medição obtida se faz presente na Tabela 16.

Os resultados compilados foram sintetizados para melhor visualização na Figura 12, onde o tempo de execução de alguns dos algoritmos analisados se sobrepõem, e na Figura 13 onde é possível observar com mais clareza o custo computacional dos algoritmos nos quais os dados coletados são similares.

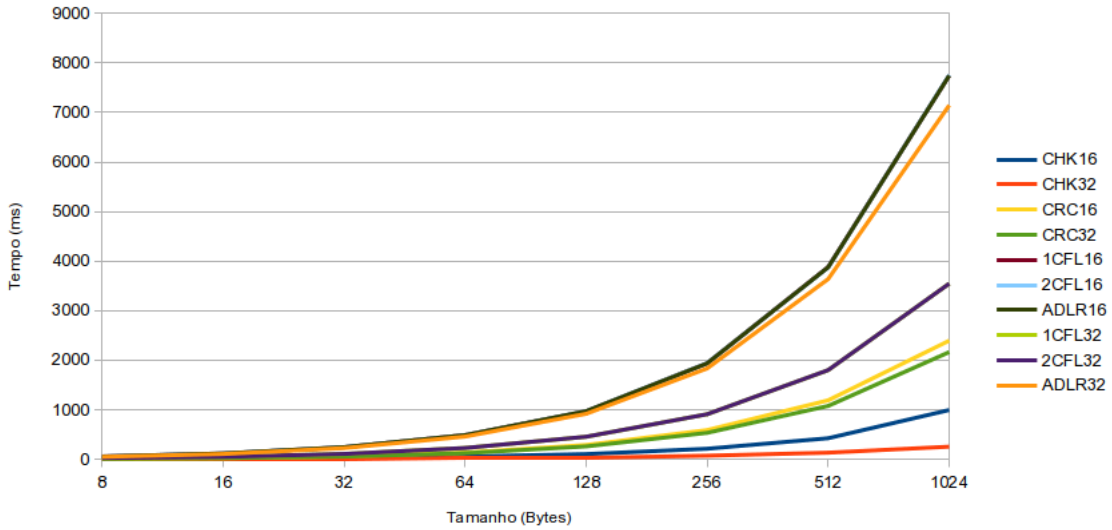
Diante dos dados dispostos, observa-se que a implementação de 32 bits do *Checksum* convencional possui custo computacional menor que sua implementação de 16 bits. Isto ocorre porque a complexidade dos algoritmos é diretamente relacionada à quantidade de blocos

Tabela 16 – Tempo de execução (milissegundos) de um milhão de cômputos de chave de verificação para cada algoritmo

N(B)	CHK16	CHK32	CRC16	CRC32	1CFL16	2CFL16	ADLR16	1CFL32	2CFL32	ADLR32
8	10	7	9	9	57	56	56	23	23	48
16	18	9	23	20	117	118	118	48	48	106
32	29	12	61	53	243	245	245	108	107	226
64	55	20	135	123	488	486	488	228	228	458
128	104	40	289	259	967	970	970	454	455	918
256	212	72	589	533	1937	1941	1936	909	909	1834
512	424	133	1191	1076	3883	3874	3877	1799	1800	3638
1024	994	254	2395	2164	7748	7757	7746	3546	3546	7147

Fonte: Produzido pelo autor

Figura 12 – Tempos de execução de um milhão de cômputos de chave de verificação para cada algoritmo em pacotes

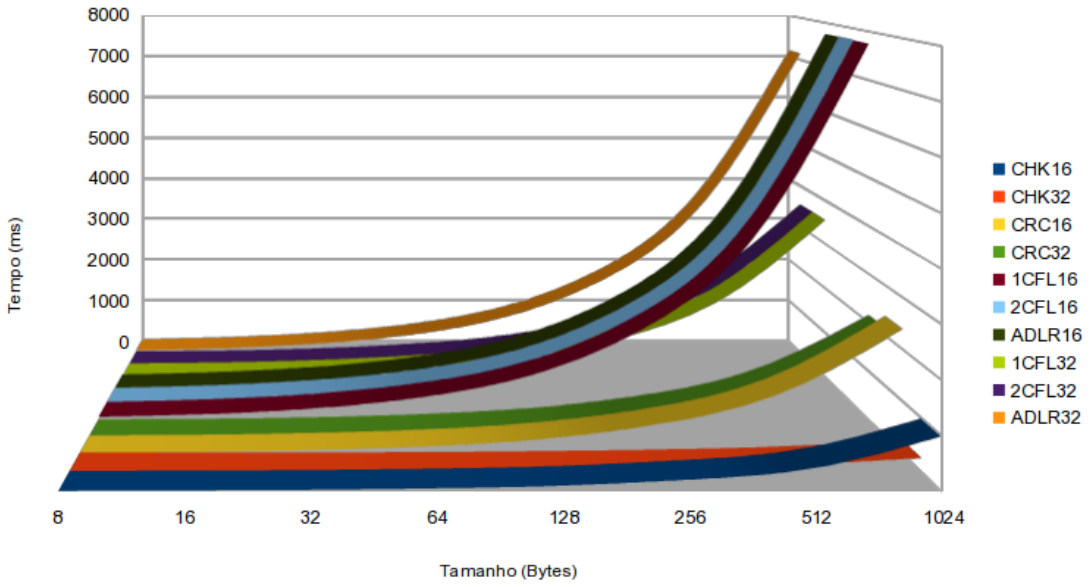


Fonte: Produzido pelo autor

processados. No algoritmo com palavra de verificação de 16 bits o pacote é fragmentado em blocos de 16 bits, já no algoritmo com palavra de verificação de 32 bits os blocos possuem 32 bits de tamanho. Logo, o segundo necessita realizar apenas metade do número de somas consecutivas que o algoritmo de 16 bits realiza. Isso demonstra que a economia de bits que compõem o pacote resulta em um custo computacional maior por parte do dispositivo que computa a chave de verificação.

Embora este fenômeno ocorra com o algoritmo de *Checksum*, o mesmo não ocorre com os algoritmos de CRC. Isto se deve ao fato de que para o cômputo do código de verificação os dados do pacote são divididos em blocos de 8 bits tanto na implementação de 16 bits quanto na de 32 bits, ou seja, o tamanho da tabela para ambos os algoritmos tem tamanho 2^8. Para aumentar o tamanho do bloco, deve-se aumentar o número de posições da *lookup table*, isto faria

Figura 13 – Tempos de execução de um milhão de cômputos de chave de verificação para cada algoritmo em pacotes



Fonte: Produzido pelo autor

com que mais espaço de memória seja necessário para computar a chave de verificação. Diante disso, é possível afirmar que independentemente do tamanho da chave de verificação, o custo computacional do algoritmo de CRC que utiliza *lookup tables* varia de acordo com o tamanho da tabela utilizada.

Quanto aos algoritmos de Fletcher e Adler, é possível observar que possuem tempos de execução muito similares quando comparados entre si. Neles, o mesmo fenômeno ocorrido com o *Checksum* convencional acontece, o tempo necessário para computar sua chave de verificação diminui conforme o tamanho dos blocos aumenta. É possível observar que o algoritmo de Adler de 32 bits é uma exceção à esta regra. Isto ocorre porque conforme dito na Subseção 2.1.4, mesmo que c_1 e c_2 tenham 16 bits de tamanho o tamanho dos blocos de dados é de 8 bits, fazendo com que o algoritmo seja cerca de duas vezes mais lendo que o *Checksum* de Fletcher. Claro que esta vantagem do algoritmo de Fletcher sobre o de Adler ocorre apenas em dados alinhados na memória em blocos de 16 bits, em dados alinhados em blocos de 8 bits os dois algoritmos teriam desempenho similar, visto que para o cômputo da chave de verificação para o algoritmo de Fletcher seria necessário que os blocos fossem percorridos a cada 8 bits, sendo concatenados dois a dois.

4.6 Combinações dois a dois

Diante dos resultados apresentados na Seção 4.5, é observado que as implementações do *Checksum* convencional apresentam tempo de execução muito inferior em comparação aos demais algoritmos (CRC, *Checksum* de Fletcher e *Checksum* de Adler). Sendo assim, faz-se lógico que seja comparada a eficácia do *Checksum* convencional com estes algoritmos mais caros porém mais robustos.

4.6.1 Algoritmos de 16 bits

Ao combinar o algoritmo de *Checksum* convencional com o de CRC, é possível observar na Tabela 17 que um algoritmo detecta a falha do outro. No cenário em que o modelo de Bernoulli é empregado, o algoritmo de *Checksum* possui várias falhas, porém estas são detectadas pelo algoritmo de CRC. Por outro lado, no cenário em que o modelo de Rajadas Periódicas é utilizado ambos possuem falhas, porém nenhuma delas passa despercebida pelos dois algoritmos, indicando que o uso desses dois algoritmos em conjunto pode ser adequado.

Tabela 17 – Eficácia do Algoritmo de *Checksum* combinado ao de CRC de 16 bits

	BER = 0.00010					T: [32, 64] | F: [8, 16]				
N(B)	f(bit)	%	CHK16	CRC16	BOTH	f(bit)	%	CHK16	CRC16	BOTH
8	1.00	1.56	2	0	0	21.54	33.65	28	0	0
16	1.00	0.78	4	0	0	36.82	28.77	15	17	0
32	1.00	0.39	12	0	0	68.67	26.83	10	8	0
64	1.00	0.20	43	0	0	132.68	25.91	15	12	0
128	1.01	0.10	140	0	0	260.68	25.46	18	10	0
256	1.02	0.05	499	0	0	516.66	25.23	14	14	0
512	1.07	0.03	1770	0	0	1028.65	25.11	15	15	0
1024	1.26	0.02	4859	1	0	2052.60	25.06	15	13	0

Fonte: Produzido pelo autor

Na Tabela 18 é possível identificar que existem falhas de transmissão que podem passar despercebidas tanto pelo algoritmo de *Checksum* quanto pelo algoritmo de Fletcher. Isto ocorre em razão da natureza dos dois algoritmos ser a mesma, somas sucessivas. Embora o algoritmo de Fletcher possua uma capacidade de detecção maior, ainda há brechas que fazem com que o uso destes algoritmos em conjunto não seja muito adequado. Este comportamento também pode ser observado. O mesmo comportamento descrito anteriormente também pode ser notado ao combinar o *Checksum* convencional com o algoritmo de Fletcher com soma em complemento de dois e o algoritmo de Adler como é exposto na Tabela 19.

Diante dos resultados obtidos na Subseção 4.6.1, é possível ainda comparar o uso do CRC de 32 bits com o uso conjunto dos algoritmos de CRC e *Checksum*, ambos com palavras de verificação de 16 bits. Para este cenário, é possível afirmar que embora ambas as alternativas sejam capazes de detectar todos os cenários produzidos, o CRC de 32 bits ainda é preferível.

Conforme observado na Subseção 4.5, o custo do algoritmo de CRC de 32 bits ainda é menor que os custos do algoritmo de CRC de 16 bits somado ao custo do *Checksum* de 16 bits.

Tabela 18 – Eficácia do Algoritmo de *Checksum* combinado ao de Fletcher (complemento de um) de 16 bits

	BER = 0.00010					T: [32, 64] | F: [8, 16]				
N(B)	f(bit)	%	CHK16	1CFL16	BOTH	f(bit)	%	CHK16	1CFL16	BOTH
8	1.00	1.56	0	0	0	21.54	33.66	28	46	1
16	1.00	0.78	2	0	0	36.81	28.76	9	16	0
32	1.00	0.39	9	0	0	68.67	26.82	12	13	0
64	1.00	0.20	40	0	0	132.66	25.91	7	15	0
128	1.00	0.10	143	0	0	260.68	25.46	22	15	0
256	1.02	0.05	528	0	0	516.67	25.23	19	12	0
512	1.07	0.03	1731	6	0	1028.68	25.11	12	21	0
1024	1.26	0.02	4934	29	10	2052.62	25.06	16	12	0

Fonte: Produzido pelo autor

Tabela 19 – Eficácia do Algoritmo de *Checksum* combinado ao de Fletcher (complemento de dois) e Adler de 16 bits

	BER = 0.00010					BER = 0.00010				
N(B)	f(bit)	%	CHK16	2CFL16	BOTH	f(bit)	%	CHK16	ADLR16	BOTH
8	1.00	1.56	0	0	0	1.00	1.56	0	0	0
16	1.00	0.78	2	0	0	1.00	0.78	2	0	0
32	1.00	0.39	11	5	0	1.00	0.39	10	0	0
64	1.00	0.20	44	14	14	1.00	0.20	35	0	0
128	1.01	0.10	145	51	37	1.01	0.10	155	0	0
256	1.02	0.05	499	192	113	1.02	0.05	533	0	0
512	1.07	0.03	1770	644	429	1.07	0.03	1785	7	1
1024	1.26	0.02	4626	1736	1134	1.26	0.02	4824	46	14

Fonte: Produzido pelo autor

Sendo assim, é possível afirmar que entre as combinações de uso dos algoritmos de 16 bits analisados neste trabalho, a que mais se mostra benéfica é a combinação entre o *Checksum* convencional e o algoritmo de CRC. Embora esta combinação não seja empregada em nenhuma pilha de protocolos atual, ela foi capaz de detectar todos os erros produzidos pelo software de simulação. Os demais algoritmos não se mostram muito benéficos visto que podem falhar em detectar alguns erros e conforme observado na Seção 4.5 são mais caros computacionalmente que o CRC com implementação baseada em *lookup table*.

4.6.2 Algoritmos de 32 bits

Quanto ao uso do *Checksum* convencional em conjunto ao CRC, é possível observar na Tabela 20, que embora o Checksum possa apresentar uma quantidade elevada de falhas de detecção no cenário mais à esquerda da tabela, estas falhas são detectadas pelo algoritmo de CRC. Ainda sim, observando o cenário representado mais à direita da tabela, é possível observar que não houveram falhas de detecção tanto por parte do *Checksum* quanto por parte do CRC.

Este comportamento foi identificado nos demais cenários de teste realizados. Sendo assim, é possível afirmar que esta combinação de algoritmos de verificação não é tão benéfica, já que apenas o CRC já é capaz de detectar a grande maioria dos erros, provendo uma detecção de erros suficientemente robusta. Isto classifica o uso do *Checksum* convencional desnecessário já que resultaria apenas em mais tempo computacional gasto.

Tabela 20 – Eficácia do Algoritmo de *Checksum* combinado ao de CRC de 32 bits

	BER = 0.00010					T = [32, 64] | F = [8, 16]				
N(B)	f(bit)	%	CHK32	CRC32	BOTH	f(bit)	%	CHK32	CRC32	BOTH
8	1.00	1.56	0	0	0	21.53	33.65	6	0	0
16	1.00	0.78	2	0	0	36.82	28.77	1	0	0
32	1.00	0.39	5	0	0	68.66	26.82	0	0	0
64	1.00	0.20	16	0	0	132.68	25.91	0	0	0
128	1.01	0.10	86	0	0	260.65	25.45	0	0	0
256	1.02	0.05	296	0	0	516.63	25.23	0	0	0
512	1.07	0.03	842	0	0	1028.73	25.12	0	0	0
1024	1.26	0.02	2339	0	0	2052.67	25.06	0	0	0

Fonte: Produzido pelo autor

Na Tabela 21 é possível observar o mesmo comportamento tanto ao se comparar o *Checksum* convencional ao algoritmo de Fletcher com soma em complemento de um quanto ao se comparar ao algoritmo de Fletcher. Isto indica quem em cenários em que uma combinação do *Checksum* convencional com um destes dois algoritmos é possível, apenas implementação do *Checksum* de Fletcher ou de Adler já é suficiente, sendo a implementação do *Checksum* convencional um desperdício de poder computacional. Para evidenciar esta característica, foi evidenciado o comportamento dos algoritmos ao serem expostos ao modelo de Bernoulli com $BER = 0.0001$. Ainda sim, deve-se levar em conta que é evidenciado na Seção 4.5 que o custo computacional do algoritmo de CRC é menor que os algoritmos de Fletcher e Adler, tornando o uso do algoritmo de CRC preferível em detrimento destes algoritmos.

Nos demais cenários (evidenciados na Tabela 5) as falhas de detecção ocorreram com muito menos frequência, mas repetindo este comportamento. Em apenas um cenário houveram duas falhas de detecção por parte do algoritmo de Adler, essas falhas ocorreram ao utilizar o Modelo de Rajadas Esparsas, provavelmente um razão da característica do algoritmo de Adler em não detectar os padrões de erros que ocorrem entre 65536 (2^k) e 65521 (maior número primo menor que 2^k).

Por fim, ao analisar os resultados obtidos pelo *Checksum* de Fletcher com soma em complemento de dois expostos na Tabela 22, nota-se um desempenho muito inferior comparado aos seus similares (Fletcher com complemento de um e Adler). Curiosamente o desempenho obtido é ainda pior que o algoritmo de Adler, que segundo Mirza (2016) deveria desempenho inferior aos algoritmos de Fletcher. Este fenômeno foi observado ao aplicar o Modelo de Bernoulli com $BER = 0.001$ nos pacotes produzidos, nos demais cenários (focados em produzir rajadas) o algoritmo de Fletcher com soma em complemento de dois não falhou nenhuma vez em detectar

Tabela 21 – Eficácia do Algoritmo de *Checksum* combinado com o de Fletcher (complemento de um) e Adler de 32 bits

	BER = 0.00010					BER = 0.00010				
N(B)	f(bit)	%	CHK32	1CFL32	BOTH	f(bit)	%	CHK32	ADLR32	BOTH
8	1.00	1.56	0	0	0	21.53	33.65	0	0	0
16	1.00	0.78	0	0	0	36.82	28.77	1	0	0
32	1.00	0.39	2	0	0	68.66	26.82	3	0	0
64	1.00	0.20	19	0	0	132.68	25.91	21	0	0
128	1.01	0.10	60	0	0	260.65	25.45	59	0	0
256	1.02	0.05	269	0	0	516.63	25.23	246	0	0
512	1.07	0.03	845	0	0	1028.73	25.12	908	0	0
1024	1.26	0.02	2365	0	0	2052.67	25.06	2339	0	0

Fonte: Produzido pelo autor

os erros injetados.

Tabela 22 – Eficácia do Algoritmo de *Checksum* combinado com o de Fletcher (complemento de dois) de 32 bits

	BER = 0.00010				
N(B)	f(bit)	%	CHK32	2CFL32	BOTH
8	1.00	1.56	0	0	0
16	1.00	0.78	1	0	0
32	1.00	0.39	1	0	0
64	1.00	0.20	29	5	5
128	1.01	0.10	76	14	9
256	1.02	0.05	289	52	37
512	1.07	0.03	885	164	113
1024	1.26	0.02	2412	465	299

Fonte: Produzido pelo autor

5
Conclusão

O objetivo deste trabalho foi o de apresentar, com o auxílio de um simulador, uma análise tanto da eficácia quanto do tempo computacional dos mecanismos de detecção de erros implementados na pilha de protocolos da Internet utilizando modelos de injeção de erros, que representam falhas de transmissão nos canais físicos. Uma breve discussão sobre os resultados finais obtidos é apresentada a seguir.

Diante dos resultados obtidos pelos testes realizados, é possível identificar cenários em que o uso de alguns algoritmos de detecção são mais adequados para uso do que outros. Além disso, foi possível analisar a utilidade do simulador de envio de pacotes em relação à sua capacidade de avaliar diferentes algoritmos de detecção de erros.

Em relação à eficácia de cada algoritmo de detecção de erros, foi possível identificar que o CRC de 32 bits tem uma capacidade de detecção de erros excelente, possuindo um custo computacional ainda menor que os Algoritmos de Fletcher e Adler. Os CRCs baseados em polinômios de 16 bits, não tiveram desempenho tão bom quanto sua versão de 32 bits e apresentaram custo computacional similar, mas ainda sim se mostra a melhor alternativa nos cenários em que o tamanho do pacote deve ser mínimo e não há condições de utilizar uma chave de verificação de 32 bits de tamanho.

Em contraste ao CRC, o algoritmo de *Checksum* não possui um desempenho tão satisfatório, principalmente nos cenários em que erros esparsos são produzidos pelos modelos de erros. Entretanto, seu tempo de execução é muito menor que qualquer outro algoritmo, diminuindo ainda mais conforme o tamanho de sua chave de verificação aumenta. Isto indica que o *Checksum* convencional se faz muito adequado para aplicações em que falhas de detecção de erros não são tão críticas e envio e recebimento rápido dos pacotes é um requisito de grande importância. Quanto aos diferentes polinômios geradores utilizados, foi notado que embora cada polinômio falhe em detectar padrões diferentes de erros, a quantidade de falhas é muito similar. Isto indica que para a escolha de $G(x)$ é necessário tomar conhecimento de que tipo de padrões

de erros ocorrem no canal de transmissão e que polinômio é capaz de detectá-los.

Embora o CRC e o *Checksum* possuem cenários bem definidos em que são mais adequados para uso, isto não fica tão claro para os algoritmos de Fletcher e Adler, que são computacionalmente mais caros e mais propensos a erros do que o algoritmo de CRC. Um possível cenário em que estes algoritmos possam ser empregados é onde se deseja ter uma capacidade de detecção de erros superior à apresentada pelo *Checksum* convencional, não há memória suficiente disponível para armazenar uma *lookup table* e não se deseja um custo computacional tão grande quanto o custo do algoritmo de CRC sem o uso de *lookup table*.

Quanto ao uso das diferentes implementações do algoritmo de Fletcher, percebeu-se que a implementação que utiliza somas em complemento de dois apresentou falhas de detecção em certos cenários. Isto ocorre possivelmente por conta da ausência da soma do bit de *carry* do bit mais significativo. Dado que ambas implementações possuem tempos de execução equivalentes, pode-se afirmar que a implementação que utiliza complemento de um é superior à implementação que utiliza complemento de dois. O *Checksum* de Adler se mostrou tão capaz de detectar erros quanto o *Checksum* de Fletcher utilizando soma em complemento de um. Entretanto, por sempre processar os dados do pacote em blocos de 8 bits, sua implementação que produz chaves de verificação de 32 bits se mostrou duas vezes mais lenta do que o algoritmo de Fletcher. O uso do algoritmo de Adler de 32 bits poderia ser possivelmente benéfico em situações em que os dados estejam alinhados a cada 8 bits em memória. Neste cenário enquanto o algoritmo de Fletcher necessitaria concatenar duas posições de memória antes de somar aos acumuladores, o algoritmo de Adler naturalmente somaria os 8 bits. Este cenário faria com que os tempos de execução dos dois algoritmos fossem muito similares.

Ao analisar os resultados obtidos ao combinar o *Checksum* convencional com os demais algoritmos de verificação foi notado que o uso de dois algoritmos em conjunto se fez benéfico apenas para os algoritmos de detecção de erros que produzem chaves de verificação de 16 bits. Para os algoritmos de 32 bits, percebe-se que ocorrem dois possíveis casos, ou apenas um algoritmo já é capaz de detectar os erros de transmissão de forma satisfatória (CRC, Fletcher com complemento de um e Adler), ou nenhum dos dois algoritmos é capaz de detectar os erros de forma satisfatória (Fletcher com complemento de dois).

Quanto ao uso do Modelo de Rajadas Esparsas, proposto como alternativa aos modelos de Gilbert e Periódico, pôde-se notar que mediante utilização deste modelo, foram criados cenários em que os algoritmos de verificação de erros apresentaram comportamentos diferentes dos apresentados com os outros modelos. Isto indica que o modelo possui grande valor para a avaliação da eficácia de algoritmos de detecção de erros, já que simula cenários que os demais modelos não foram capazes de gerar.

Por fim, é possível afirmar que o simulador construído serviu de grande ajuda para melhor avaliar a capacidade de detecção de erros e custo computacional dos algoritmos de detecção de erros. Foi notado que realizar a implementação das versões mais otimizadas de cada algoritmo é

uma tarefa extremamente árdua e que algumas vezes podia até mesmo depender do compilador ou da arquitetura do computador utilizado para realizar os testes. Felizmente, o simulador é completamente escalável e de código aberto, o que permite que quaisquer outros algoritmos de verificação tenham duas capacidades de verificação medidas de acordo com os modelos de injeção de erros e rotinas de teste dispostas.

Diante do resultados obtidos, têm-se como trabalhos futuros:

a) Implementar versões ainda mais otimizadas dos algoritmos de verificação;

b) Analisar a eficácia de funções *hash* como alternativa aos algoritmos de detecção convencionais;

c) Tornar o simulador capaz de detectar padrões específicos de erros de transmissão que levam à falha de detecção por meio de documentos de *logs*;

Referências

AZAHARI, A. et al. Review of error detection of data link layer in computer network. *ARPN J. Eng. and Applied Sciences*, Asian Research Publishing Network (ARPN), v. 9, n. 1, p. 1–4, 1 2014. ISSN 1819-6608. Citado 3 vezes nas páginas 15, 20 e 22.

DEUTSCH, L. P. *DEFLATE Compressed Data Format Specification version 1.3*. RFC Editor, 1996. RFC 1951. (Request for Comments, 1951). Disponível em: <https://www.rfc-editor.org/info/rfc1951>. Acesso em: 04 jul 2022. Citado na página 15.

DEUTSCH, P.; GAILLY, J.-L. *ZLIB Compressed Data Format Specification version 3.3*. RFC Editor, 1996. RFC 1950. (Request for Comments, 1950). Disponível em: <https://www.rfc-editor.org/info/rfc1950>. Acesso em: 04 jul 2022. Citado 2 vezes nas páginas 19 e 23.

FLETCHER, J. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, v. 30, n. 1, p. 247–252, 1982. Citado 3 vezes nas páginas 15, 21 e 22.

JIANG, W.; SCHULZRINNE, H. Modeling of packet loss and delay and their effect on real-time multimedia service quality. In: *PROCEEDINGS OF NOSSDAV '2000*. [S.l.: s.n.], 2000. Citado na página 24.

KOOPMAN, P. *Best CRC Polynomials*. 2015. Disponível em: <https://users.ece.cmu.edu/~koopman/crc/>. Acesso em: 04 jul 2022. Citado 2 vezes nas páginas 26 e 33.

KOOPMAN, P.; DRISCOLL, K. R.; HALL, B. *Selection of Cyclic Redundancy Code and Checksum Algorithms to Ensure Critical Data Integrity*. [S.l.], 2015. Citado na página 15.

KUROSE, J. F.; ROSS, K. W. *Redes de Computadores e a Interent: uma abordagem top-Down*. 6. ed. [S.l.]: Pearson, 2012. ISBN 0132856204, 9780132856201. Citado 2 vezes nas páginas 15 e 16.

MAXINO, T. C. Revisiting fletcher and adler checksums. In: . [S.l.: s.n.], 2006. Citado na página 22.

MAXINO, T. C.; KOOPMAN, P. J. The effectiveness of checksums for embedded control networks. *IEEE Trans. on Dependable and Secure Computing*, v. 6, n. 1, p. 59–72, Jan 2009. ISSN 1545-5971. Citado 3 vezes nas páginas 15, 16 e 23.

MCCOY, W. *Implementation guide for the ISO Transport Protocol*. RFC Editor, 1987. RFC 1008. (Request for Comments, 1008). Disponível em: <https://www.rfc-editor.org/info/rfc1008>. Acesso em: 04 jul 2022. Citado na página 19.

MIRZA, A. N. *Analyzing error detection performance of checksums in embedded networks*. Dissertação (Mestrado) — Strathmore University, 2016. Citado 3 vezes nas páginas 23, 24 e 45.

NAKASSIS, A. Fletcher's error detection algorithm: How to implement it efficiently and how toavoid the most common pitfalls. *SIGCOMM Comput. Commun. Rev.*, Association for Computing Machinery, New York, NY, USA, v. 18, n. 5, p. 63–88, oct 1988. ISSN 0146-4833. Disponível em: <https://doi.org/10.1145/53644.53648>. Acesso em: 04 jul 2022. Citado 2 vezes nas páginas 22 e 23.

NEGRIZOLI, I.; RODRIGUES, L.; OYAMADA, M. Uma análise da eficácia dos mecanismos de detecção de erros da pilha de protocolos da internet. In: *Anais do XXII Workshop de Testes e Tolerância a Falhas*. Porto Alegre, RS, Brasil: SBC, 2021. p. 71–84. ISSN 2595-2684. Disponível em: <https://sol.sbc.org.br/index.php/wtf/article/view/17205>. Acesso em: 04 jul 2022. Citado 10 vezes nas páginas 16, 20, 21, 24, 25, 26, 28, 29, 53 e 54.

PANDURANGAN, V. *Linux kernel bug delivers corrupt TCP/IP data to Mesos, Kubernetes, Docker containers*. 2016. Https://tech.vijayp.ca/linux-kernel-bug-delivers-corrupt-tcp-ip-data-to-mesos-kubernetes-docker-containers-4986f88f7a19. Acesso em: 04 jul 2022. Citado na página 16.

PETERSON, W. W.; BROWN, D. T. Cyclic codes for error detection. *Proceedings of the IRE*, v. 49, n. 1, p. 228–235, Jan 1961. ISSN 0096-8390. Citado na página 20.

PLUMMER, W. W. *Computing the Internet checksum*. RFC Editor, 1988. RFC 1071. (Request for Comments, 1071). Disponível em: <https://www.rfc-editor.org/info/rfc1071>. Acesso em: 04 jul 2022. Citado 3 vezes nas páginas 18, 19 e 20.

POHORESKI, M. *CRC32 Demystified*. [S.l.]: GitHub, 2017. <https://github.com/Michaelangel007/crc32>. Acesso em: 04 jul 2022. Citado na página 29.

RAHMANI, M. et al. Error detection capabilities of automotive network technologies and ethernet - a comparative study. In: *2007 IEEE Intelligent Vehicles Symposium*. [S.l.: s.n.], 2007. p. 674–679. ISSN 1931-0587. Citado na página 22.

SHEINWALD, D. et al. *Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations*. RFC Editor, 2002. RFC 3385. (Request for Comments, 3385). Disponível em: <https://www.rfc-editor.org/info/rfc3385>. Acesso em: 04 jul 2022. Citado na página 26.

SOUSA, P. B.; FERREIRA, L. L. *Bit Error Models*. [S.l.], 2007. Citado na página 26.

STANCLIFF, M. *crcspeed*. [S.l.]: GitHub, 2020. <https://github.com/mattsta/crcspeed>. Acesso em: 04 jul 2022. Citado na página 29.

STONE, J. et al. Performance of checksums and CRC's over real data. *IEEE/ACM Trans. Netw.*, IEEE Press, Piscataway, NJ, USA, v. 6, n. 5, p. 529–543, Oct 1998. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109/90.731187>. Acesso em: 04 jul 2022. Citado 3 vezes nas páginas 16, 20 e 22.

TOURNOUX, P. U. et al. On-the-fly erasure coding for real-time video applications. *IEEE Transactions on Multimedia*, v. 13, n. 4, p. 797–812, 2011. Citado na página 24.

WOLF, J. K.; BLAKENEY, R. D. An exact evaluation of the probability of undetected error for certain shortened binary CRC codes. In: *MILCOM 88, 21st Military Communications Conference*. [S.l.: s.n.], 1988. p. 287–292 vol.1. Citado na página 16.

ZWEIG, J.; PARTRIDGE, D. C. *TCP alternate checksum options*. RFC Editor, 1990. RFC 1146. (Request for Comments, 1146). Disponível em: <https://www.rfc-editor.org/info/rfc1146>. Acesso em: 04 jul 2022. Citado na página 15.

Apêndices

APÊNDICE A

Diagrama de classes antes das mudanças

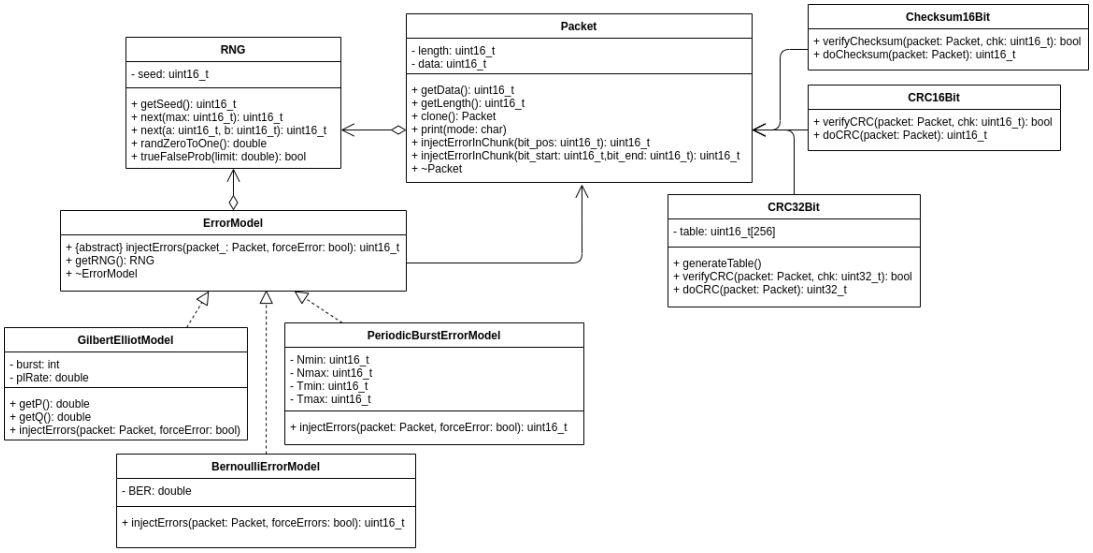
Na Figura 14 está disposto o diagrama de classes, que ilustra o comportamento do simulador durante sua execução.

A classe *RNG* é responsável por gerar números pseudo-aleatórios que serão atribuídos ao conteúdo (*payload*) dos pacotes, bem como nos modelos de injeção de erros. Para tanto, ela utiliza a biblioteca *random* [1] da linguagem C++. Uma semente inicial pode ser definida para permitir a reprodutibilidade dos experimentos.

Após a construção do pacote com os dados gerados, o cálculo de seu código de verificação é feito de acordo com os diferentes algoritmos implementados, como é feito pelo processo emissor. Em seguida, os erros de transmissão são injetados no pacote pelo modelo de erros instanciado (classes que implementam $ErrorModel$). Por fim, é feito novamente o cálculo dos códigos de verificação e comparado com o anterior, simulando a recepção do pacote no destinatário. Para que ao menos um erro de transmissão seja injetado em cada pacote (modelos com taxas de erro muito baixas podem não injetar erros em alguns pacotes), um parâmetro $forceError = true$ pode ser utilizado para gerar ao menos um erro aleatório quando nenhum bit é modificado diretamente pelo modelo. Neste caso, como todos os pacotes terão ao menos um erro de bit, se o valor obtido pelo método de verificação é diferente, significa que o algoritmo obteve êxito em detectar o erro. Caso contrário, significa que o algoritmo de detecção de erros falhou em detectar o erro de transmissão injetado. Esse processo é repetido várias vezes e dados estatísticos referentes à eficácia dos algoritmos são gerados como pode ser observado nas tabelas expostas nesta seção. (NEGRIZOLI; RODRIGUES; OYAMADA, 2021)

[1] <https://www.cplusplus.com/reference/random/>

Figura 14 – Diagrama de classes do PacketSenderSim



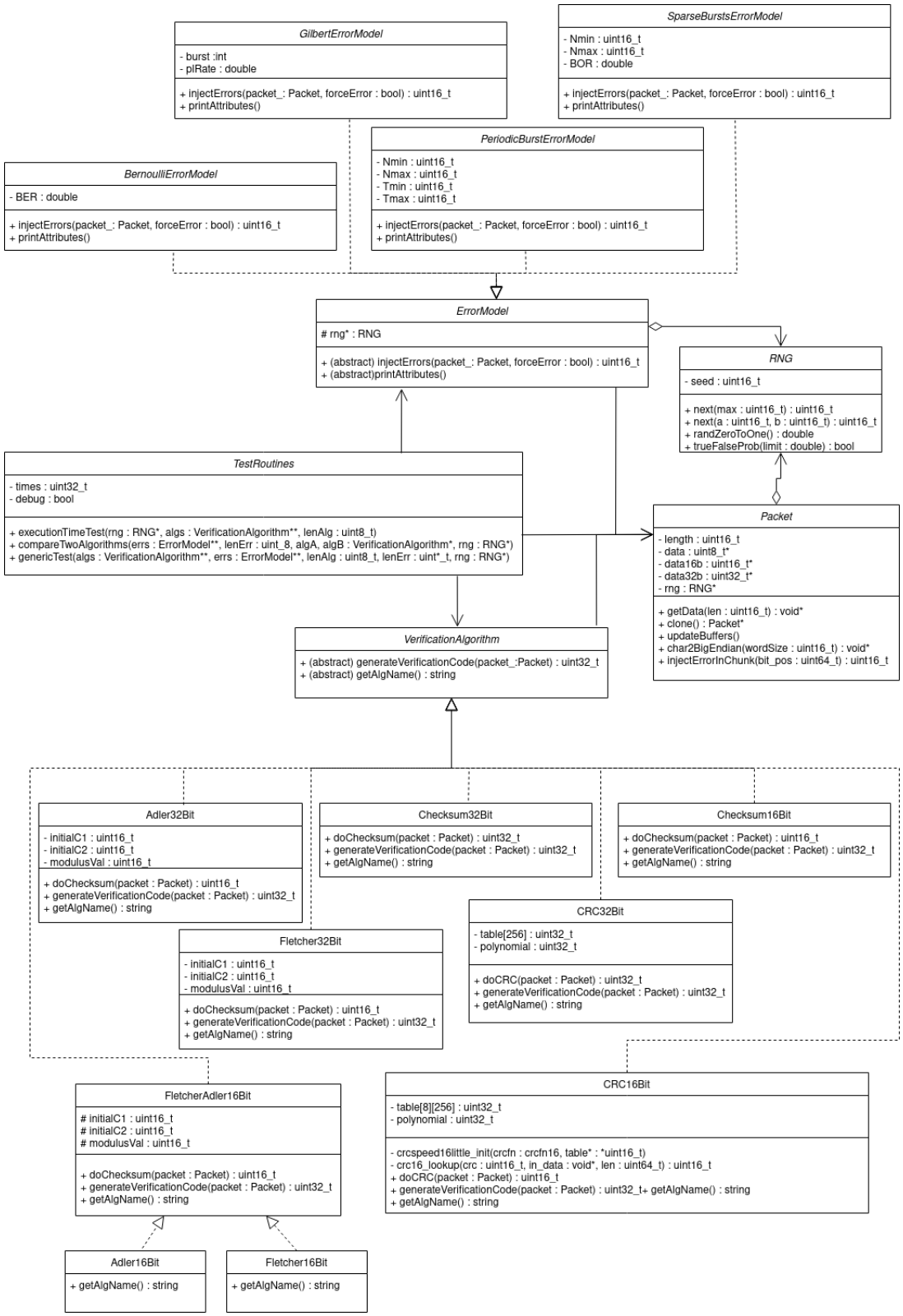
Fonte: Negrizoli, Rodrigues e Oyamada (2021)

APÊNDICE B

Diagrama de classes após mudanças

Na Figura 15 está disposto o diagrama de classes após as mudanças descritas na Seção 3.1 serem aplicas ao simulador e submetidas ao repositório. Este diagrama de classes, assim como o anterior, segue o padrão UML.

Figura 15 – Diagrama de classes do PacketSenderSim



Fonte: Produzido pelo autor

APÊNDICE C

Código exemplo

Para utilizar o *framework*, primeiramente se faz a importação do cabeçalho "packetSenderSim.hpp". Este cabeçalho condensa todas as importações dos modelos de erros, algoritmos de verificação e rotinas de teste. No código exemplo também se fazem presentes algumas definições de constantes para melhorar a leiturabilidade dos próximos trechos de código.

Como requisito para a execução da rotina de testes, devem ser instanciados previamente o gerador de números aleatórios (RNG), um vetor unidimensional para armazenar as posições de memória dos algoritmos de verificação que terão suas eficácias testadas (VerificationAlgorithm) e outro vetor unidimensional para armazenar as posições de memória dos modelos de erros que serão utilizados para pôr os algoritmos de verificação à prova (ErrorModel). Após isso, declara-se dinamicamente os algoritmos de detecção e os modelos de injeção de erros. E por fim, após as instanciações, é possível executar as rotinas de teste de acordo com os algoritmos de detecção e modelos de erros instanciados previamente.

Código 1 – Código com exemplo de utilização do *framework*

```
#include "packetSenderSim.hpp"
#include <iostream>
#define DEBUG false
#define TIMES 1000000
#define SEED 0x1234

int main(){
  RNG* rng = new RNG(SEED);
  VerificationAlgorithm *algs[10];
  ErrorModel *errs[10];
  ErrorModel *errs2[5];

  algs[0] = new Checksum16Bit();
  algs[1] = new Checksum32Bit();
  algs[2] = new CRC16Bit(0x1021);
  algs[3] = new CRC32Bit(0x04C11DB7);
  algs[4] = new Fletcher16Bit(true);
  algs[5] = new Fletcher16Bit(false);
  algs[6] = new Adler16Bit();
  algs[7] = new Fletcher32Bit(true);
  algs[8] = new Fletcher32Bit(false);
  algs[9] = new Adler32Bit();

  errs[0] = new SparseBurstsErrorModel(0.0001, 16, 32, rng);
  errs[1] = new SparseBurstsErrorModel(0.0001, 32, 64, rng);
  errs[2] = new SparseBurstsErrorModel(0.0001, 32, 64, rng);
  errs[3] = new BernoulliErrorModel(0.001, rng);
  errs[4] = new BernoulliErrorModel(0.0001, rng);
  errs[5] = new GilbertErrorModel(2, 0.001, rng);
  errs[6] = new GilbertErrorModel(2, 0.0001, rng);
  errs[7] = new GilbertErrorModel(8, 0.0001, rng);
  errs[8] = new PeriodicBurstErrorModel(1,16, rng);
  errs[9] = new PeriodicBurstErrorModel(8,16,32,64, rng);

  errs2[0] = new SparseBurstsErrorModel(0.0001, 32, 64, rng);
  errs2[1] = new BernoulliErrorModel(0.0001, rng);
  errs2[2] = new GilbertErrorModel(8, 0.0001, rng);
  errs2[3] = new GilbertErrorModel(16, 0.0001, rng);
  errs2[4] = new PeriodicBurstErrorModel(8,16,32,64, rng);

  TestRoutines* test = new TestRoutines(TIMES, DEBUG);

  test->genericTest(algs, errs, 10, 10, rng);
  test->executionTimeTest(rng, algs, 10);
  std::cout << std::endl << std::endl;
  test->compareTwoAlgorithms(errs2, 5, algs[1], algs[2], rng);
  std::cout << std::endl << std::endl;
  test->compareTwoAlgorithms(errs2, 5, algs[1], algs[3], rng);
  std::cout << std::endl << std::endl;
  test->compareTwoAlgorithms(errs2, 5, algs[1], algs[5], rng);
  std::cout << std::endl << std::endl;
  test->compareTwoAlgorithms(errs2, 5, algs[1], algs[7], rng);

  delete rng;
  return 0;
}
```