UNIVERSIDAD DEL VALLE DE GUATEMALA
Redes - CC3067
Sección 11
Ing. Miguel Novella Linares

Laboratorio 2

Esquemas de detección y corrección

José Pablo Orellana 21970
Diego Alberto Leiva 21752

GUATEMALA, 24 de julio del 2024

Esquema de Corrección de Errores Hamming

Escenarios de Prueba

Sin errores

```
C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>python emisor.py
Ingrese un mensaje en binario: 1001
Mensaje codificado: 0011001

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>g++ receptor.cpp -o receptor

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>receptor
Ingrese un mensaje codificado en binario: 0011001
No se detectaron errores. Mensaje original: 0011001
```

```
C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>python emisor.py
Ingrese un mensaje en binario: 110011
Mensaje codificado: 1011100011

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>g++ receptor.cpp -o receptor

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>receptor
Ingrese un mensaje codificado en binario: 1011100011
No se detectaron errores. Mensaje original: 1011100011
```

```
C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2>cd Hamming

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>python emisor.py
Ingrese un mensaje en binario: 1000001
Mensaje codificado: 00100001001

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>g++ receptor.cpp -o receptor

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>receptor
Ingrese un mensaje codificado en binario: 00100001001
No se detectaron errores. Mensaje original: 00100001001
```

Un error

```
C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>python emisor.py
Ingrese un mensaje en binario: 1001
Mensaje codificado: 0011001

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>g++ receptor.cpp -o receptor

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>receptor
Ingrese un mensaje codificado en binario: 0001001
Error detectado en la posicion: 3
Mensaje corregido: 0011001
```

```
C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>python emisor.py
Ingrese un mensaje en binario: 110011
Mensaje codificado: 1011100011

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>g++ receptor.cpp -o receptor

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>receptor
Ingrese un mensaje codificado en binario: 1011110011
Error detectado en la posicion: 6
Mensaje corregido: 1011100011
```

```
C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>python emisor.py
Ingrese un mensaje en binario: 1000001
Mensaje codificado: 00100001001

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>g++ receptor.cpp -o receptor

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>receptor
Ingrese un mensaje codificado en binario: 10100001001
Error detectado en la posicion: 1
Mensaje corregido: 00100001001
```

Dos o más errores

```
C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>python emisor.py
Ingrese un mensaje en binario: 1001
Mensaje codificado: 0011001

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>g++ receptor.cpp -o receptor

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>receptor
Ingrese un mensaje codificado en binario: 0000001
Error detectado en la posicion: 7
Mensaje corregido: 0000000
```

```
C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>python emisor.py
Ingrese un mensaje en binario: 110011
Mensaje codificado: 1011100011

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>g++ receptor.cpp -o receptor

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>receptor
Ingrese un mensaje codificado en binario: 1011100000
Error detectado en la posicion: 3
Mensaje corregido: 1001100000
```

```
C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>python emisor.py
Ingrese un mensaje en binario: 1000001
Mensaje codificado: 00100001001

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>g++ receptor.cpp -o receptor

C:\Users\Pablo Orellana\Documents\GitHub\Redes-Lab2\Hamming>receptor
Ingrese un mensaje codificado en binario: 11100001001
Error detectado en la posicion: 3
Mensaje corregido: 11000001001
```

Preguntas

¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error? ¿Por qué sí o por qué no? En caso afirmativo, demuestrelo con su implementación.

- Si es posible manipular los bits de manera que el algoritmo no pueda detectar el error. Los códigos de Hamming están diseñados para poder detectar y corregir errores en un sólo bit. Si tenemos más de un error, los códigos de Hamming pueden detectar que hay un error, más no pueden corregirlo de manera fiable. En el caso de dos o más errores el código puede fallar al detectar errores correctamente, incluso corregir incorrectamente el mensaje. Y todo esto se debe a que los bits de paridad calculados pueden coincidir incorrectamente con los bits de paridad esperados. Un ejemplo se muestra en la imagen anterior (Título de imagen: “Dos o más errores”).

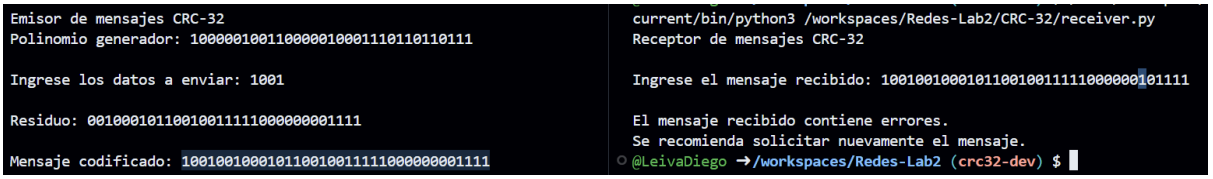
Esquema de Detección de Errores CRC-32

Escenarios de Prueba

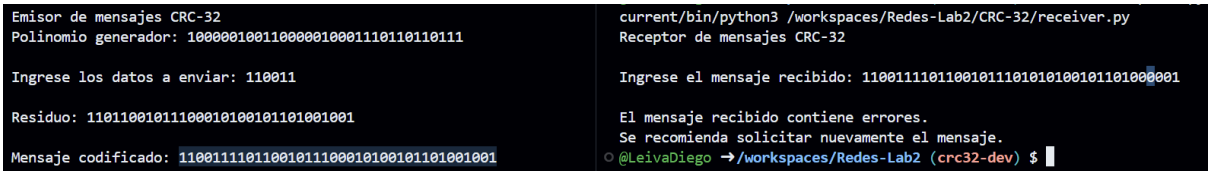
Sin errores

El CRC-32 genera correctamente el residuo de 32 bits para codificar el mensaje original

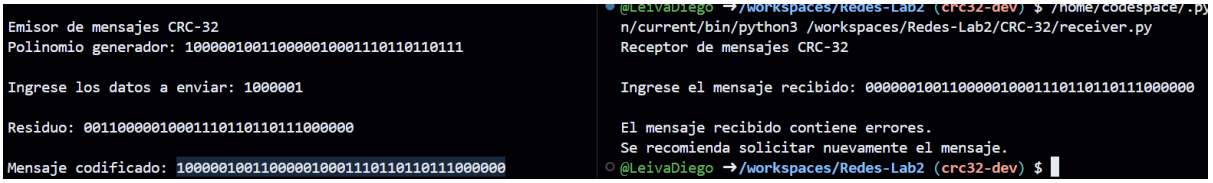
Un error



Error en bit 6 (de derecha a izquierda)



Error en bit 4 (de derecha a izquierda)

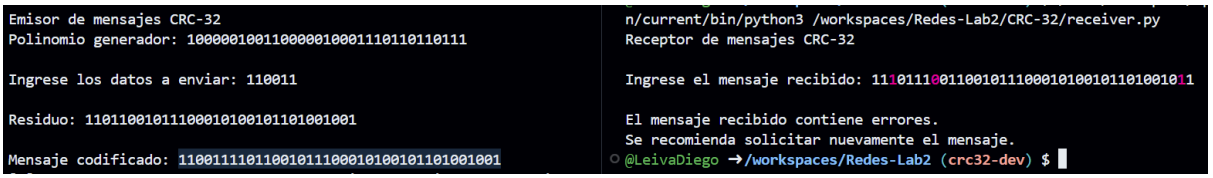


Error en bit 1 (de izquierda a derecha)

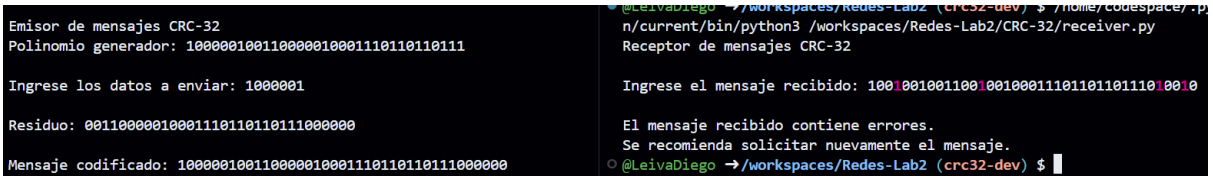
El CRC-32 es capaz de detectar que existe un error dentro del mensaje, ya sea que el error esté en el mensaje original o en el checksum generado. Sin embargo, no se tiene la capacidad de detectar en qué bit en específico esta el error.

Dos o más errores

Errores en bits 2 (de izquierda a derecha) y bit 2 (de derecha a izquierda)



Errores en bits 3 y 7 (de izquierda a derecha) y error en bit 2 (de derecha a izquierda)



Errores en bits 4 y 14 (de izquierda a derecha) y errores en bits 2 y 5 (de derecha a izquierda)

Nuevamente el CRC-32 demuestra la capacidad que tiene en la detección de múltiples errores, aunque sigue sin tener la capacidad de detectar en qué bits se encuentran.

Preguntas

¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error? ¿Por qué sí o por qué no? En caso afirmativo, demuestrelo con su implementación.

- Si es posible manipular los bits de tal manera que el CRC-32 no detecte el error. Esto es posible debido a que este algoritmo no es una función criptográfica, es decir no encripta el mensaje, solo genera un checksum a revisar posteriormente. Este algoritmo está diseñado principalmente para detectar errores aleatorios en los mensajes. Algunos post de stackoverflow, mencionan que la probabilidad de no detectar un error depende de la

cantidad de bits en error y la longitud del mensaje. Se menciona que para un CRC-16 con un patrón aleatorio puede pasar la verificación 1/65,536 veces. () Asimismo se muestran la cantidad de posibles patrones que pueden burlar la verificación.

```
48 bit data, 16 bit crc, => 64 bit message
2^64 - 1 possible error patterns
       84 of        635376 possible patterns of  4 error bits fail
     2430 of      74974368 possible patterns of  6 error bits fail
   133001 of    4426165368 possible patterns of  8 error bits fail
  4621021 of  151473214816 possible patterns of 10 error bits fail
100246083 of 3284214703056 possible patterns of 12 error bits fail
```

(Reid, 2018)
Debido a que las probabilidades de burlar la verificación son muy bajas y se requiere de patrones muy específicos no fue posible demostrarlo con la implementación actual.

En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos? Tome en cuenta complejidad, velocidad, redundancia (overhead), etc.

- Dentro de las ventajas que el algoritmo CRC-32 posee es la alta eficiencia en la detección de errores aleatorios y rafagas de errores. Asimismo es un algoritmo que es muy rápido de calcular, sobre todo cuando se implementa a nivel hardware.
- Otra ventaja que tiene es que posee la capacidad de detectar múltiples errores, cosa que Hamming no puede hacer.
- Frente a Hamming CRC32 tiene una desventaja en cuanto a overhead, pues agrega 32 bits, independientemente de que tamaño tenga el mensaje.
- Otra desventaja de CRC32 es el hecho de que no tiene la capacidad de corregir errores, solo detectarlos.
- En casos muy específicos CRC32 puede llegar a ser burlado y evitar que se detecten errores.
- Una de las ventajas que tiene el código de Hamming es la capacidad que tiene de detectar errores en múltiples bits. Y además tiene bajo Overhead ya que añade un número limitado de bits de paridad al mensaje original.
- Una de sus principales desventajas es que tiene un límite de corrección de un sólo bit y no puede corregir múltiples errores, y además tiene una complejidad que requiere de cálculos para determinar y verificar los bits de paridad.

Repositorio GitHub:
https://github.com/LeivaDiego/Redes-Lab2.git

Referencias

Aquanar. (2021). *How is a CRC32 checksum calculated?* Stack Overflow. https://stackoverflow.com/questions/2587766/how-is-a-crc32-checksum-calculated

Bies, L. (2021). *On-line CRC calculation and free library - Lammert Bies*. Lammert Bies. https://www.lammertbies.nl/comm/info/crc-calculation

CRC-32. (s. f.). https://fuchsia.googlesource.com/third_party/wuffs/+/HEAD/std/crc32/README.md

Davies, J. (2009). *Understanding CRC32*. Command Line Fanatic. https://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art008

Molkenthin, B. (2015). *Understanding CRC*. http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html#ch6

Reid, J. (2018). *What kind of errors does CRC method cannot detect?* Stack Overflow. https://stackoverflow.com/questions/52357777/what-kind-of-errors-does-crc-method-cannot-detect