Universidad del Valle de Guatemala
Redes
Sección 20

Laboratorio No. 2

Carlos Edgardo López Barrera 21666

Brian Anthony Carrillo Monzon - 21108

Guatemala, 25 de julio del 2024

Desarrollo

- Algoritmo de corrección de errores (Hamming(7,4)):

  Escenarios de pruebas :

  - Prueba 1
    - Emisor

```
carloslopez@Carloss-MacBook-Pro Redes-Labs % java Hamming
Ingrese un mensaje
1010
Mensaje codificado: 0101101
carloslopez@Carloss-MacBook-Pro Redes-Labs % java Hamming
Ingrese un mensaje
11011001
Mensaje codificado: 100101011100
carloslopez@Carloss-MacBook-Pro Redes-Labs % java Hamming
Ingrese un mensaje
010101
Mensaje codificado: 1011010000
```

    - Receptor

```
carloslopez@Carloss-MacBook-Pro Redes-Labs % /usr/bin/py
Ingrese el mensaje codificado: 0101101
Mensaje decodificado: 1010
carloslopez@Carloss-MacBook-Pro Redes-Labs % /usr/bin/py
Ingrese el mensaje codificado: 100101011100
Mensaje decodificado: 11011001
carloslopez@Carloss-MacBook-Pro Redes-Labs % /usr/bin/py
Ingrese el mensaje codificado: 1011010000
Mensaje decodificado: 010101
```

  - Prueba 2
    - Emisor

```
carloslopez@Carloss-MacBook-Pro Redes-Labs % java Hamming
Ingrese un mensaje
1011
Mensaje codificado: 1100110
carloslopez@Carloss-MacBook-Pro Redes-Labs % java Hamming
Ingrese un mensaje
11001
Mensaje codificado: 110011111
carloslopez@Carloss-MacBook-Pro Redes-Labs % java Hamming
Ingrese un mensaje
1110001
Mensaje codificado: 10010110111
```

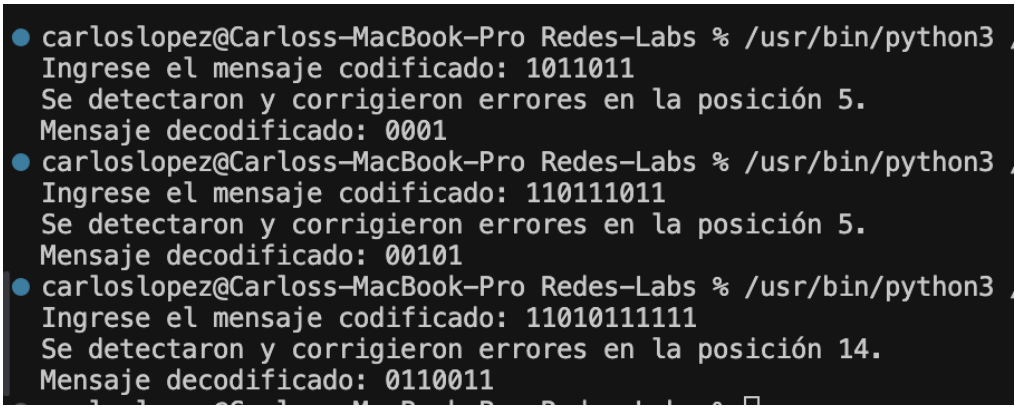
- Receptor
  - Mensajes modificados
    - 1100110 (se modificó la posición número 2 → 1 a 0)
    - 110011111 (se modificó la posición número 4 → 1 a 0)
    - 10010110111 (se modificó la posición número 10 → 0 a 1)

```
carloslopez@Carloss-MacBook-Pro Redes-Labs % /usr/bin/python3
Ingrese el mensaje codificado: 1100100
Se detectaron y corrigieron errores en la posición 2.
Mensaje decodificado: 1011
carloslopez@Carloss-MacBook-Pro Redes-Labs % /usr/bin/python3
Ingrese el mensaje codificado: 110010111
Se detectaron y corrigieron errores en la posición 4.
Mensaje decodificado: 11001
carloslopez@Carloss-MacBook-Pro Redes-Labs % /usr/bin/python3
Ingrese el mensaje codificado: 11010110111
Se detectaron y corrigieron errores en la posición 10.
Mensaje decodificado: 1110001
```

- Prueba 3
  - Emisor

```
carloslopez@Carloss-MacBook-Pro Redes-Labs % java Hamming
Ingrese un mensaje
1111
Mensaje codificado: 1111111
carloslopez@Carloss-MacBook-Pro Redes-Labs % java Hamming
Ingrese un mensaje
11001
Mensaje codificado: 110011111
carloslopez@Carloss-MacBook-Pro Redes-Labs % java Hamming
Ingrese un mensaje
1110001
Mensaje codificado: 10010110111
```

  - Receptor
    - Mensajes modificados
      - 1111111 (se modificó la posición número 5 → 1 a 0 y la posición número 3 → 1 a 0)
      - 110011111 (se modificó la posición número 5 → 1 a 0 y la posición número 2 → 1 a 0)
      - 10010110111 (se modificó la posición número 9 → 0 a 1 y la posición número 3 → 0 a 1)



¿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 el caso del Código de Hamming (7, 4), aunque es de corrección de errores, aun así es posible manipular los bits para que el algoritmo no detecte el error. Esto puede pasar cuando hay dos o más errores en el mensaje, ya que, el Código de Hamming (7, 4) está diseñado para detectar y corregir errores de un solo bit, pero no puede detectar correctamente errores múltiples.

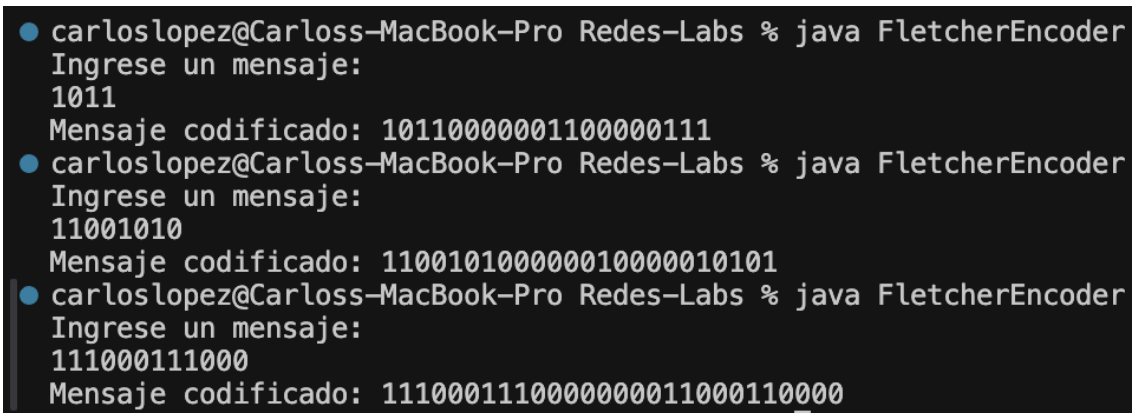
En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos?

En la implementación del Código de Hamming (7, 4), observamos que, en comparación con otros métodos de detección y corrección de errores, la redundancia de este es significativa. Esto porque, para un mensaje de 4 bits, se añaden 3 bits de paridad, resultando en un 75% de overhead. Esta redundancia puede ser una desventaja en sistemas donde el ancho de banda es limitado. Una ventaja que notamos es la capacidad de corrección de errores, aunque, solo puede corregir errores de un solo bit, lo que es muy útil en entornos donde los errores aislados son comunes, pero, cuando se trata de más de un error, este algoritmo no lo detecta. En cuanto a su implementación y cálculo de bits de paridad, son más complejos en comparación con métodos más simples, lo que se refleja en la velocidad de ejecución, que es ligeramente más lenta debido al procesamiento adicional requerido para calcular y verificar los bits de paridad.

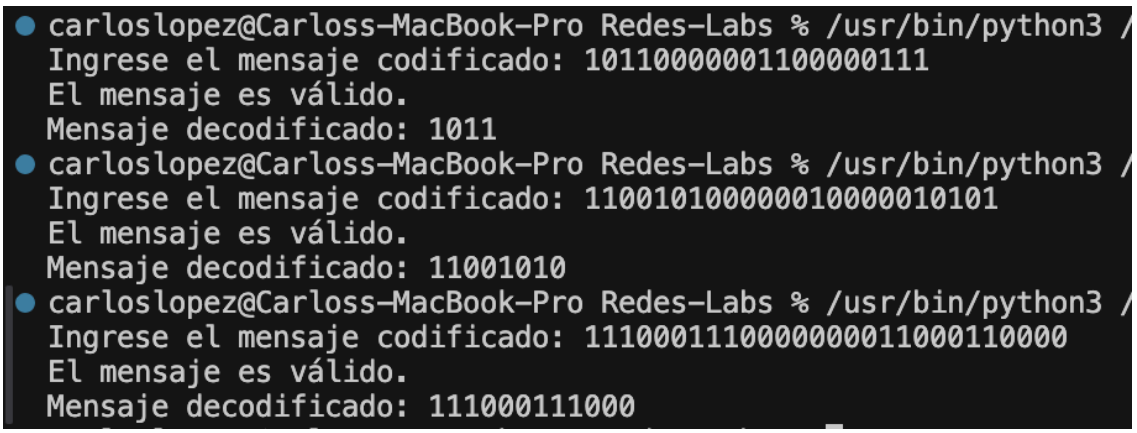
- Algoritmo de detección de errores (Fletcher Checksum):

  Escenarios de pruebas :

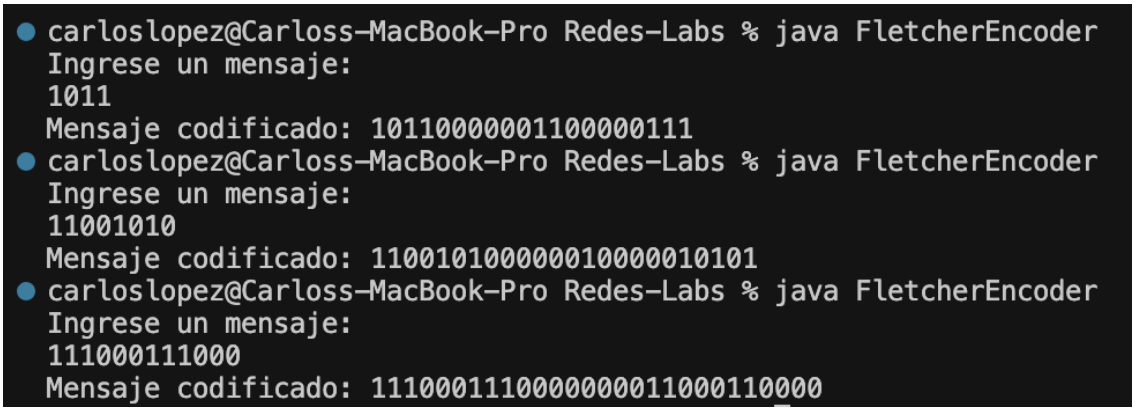
  - Prueba 1
    - Emisor



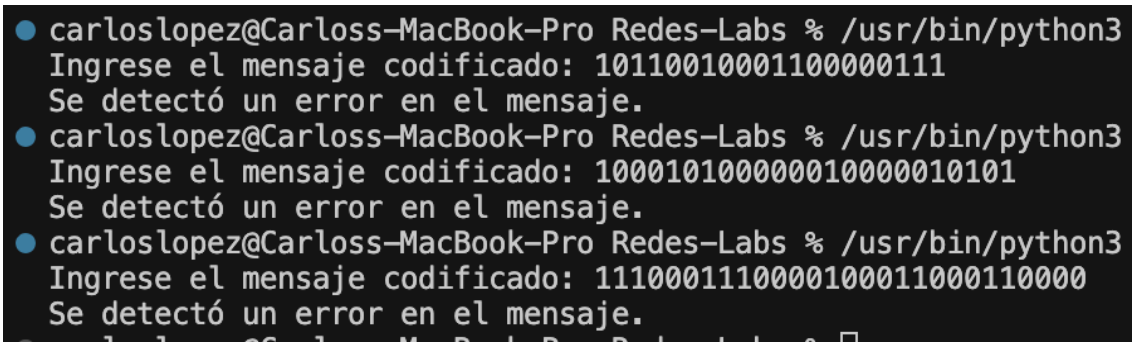
    - Receptor



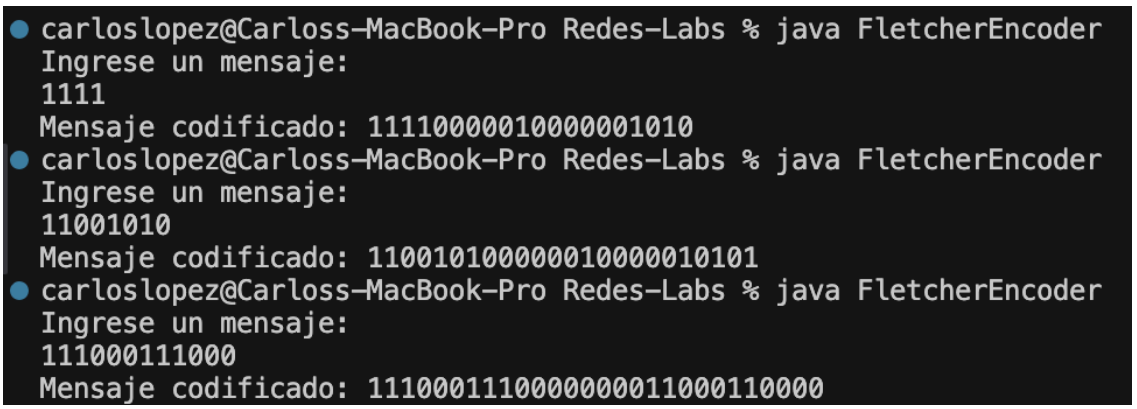
  - Prueba 2
    - Emisor



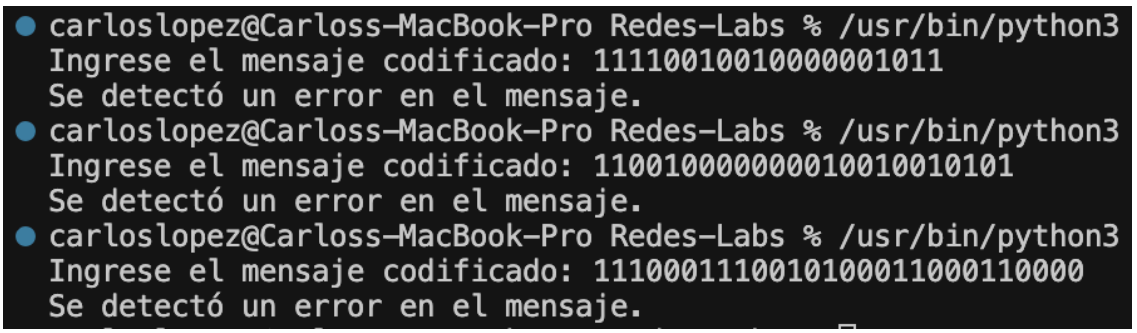
- Receptor
  - Mensajes modificados
    - 10110010001100000111 (se modificó la posición número 14 → 0 a 1)
    - 10001010000010000010101 (se modificó la posición número 23 → 1 a 0)
    - 11100011100001000110001100000 (se modificó la posición número 15 → 0 a 1)



- Prueba 3
  - Emisor



  - Receptor
    - Mensajes modificados
      - 11110010010000001011 (se modificó la posición número 1 → 0 a 1 y la posición número 14 → 0 a 1)
      - 11001010000000100100101 (se modificó la posición número 8 → 0 a 1 y la posición número 19 → 0 a 1)
      - 11100011100000000011000110000 (se modificó la posición número 15 → 0 a 1 y la posición número 17 → 0 a 1)



¿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?

Sí, es posible manipular los bits de tal manera que el algoritmo Fletcher Checksum no detecte el error, pero esto solo puede ocurrir en casos muy específicos donde los errores múltiples se compensan mutuamente de manera que el checksum resultante no cambia.

En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos?

En la implementación del algoritmo Fletcher Checksum, notamos que comparado con otros métodos, la redundancia es relativamente baja y fija. El Fletcher Checksum añade 16 bits de checksum independientemente del tamaño del mensaje, lo cual es ventajoso en términos de eficiencia, especialmente para mensajes largos. Otra ventaja que vimos, es que la implementación es bastante sencilla y rápida. Consiste en sumar los valores de los bits y aplicar módulos, lo cual es menos complejo y más veloz en comparación con algoritmos que requieren cálculos de bits de paridad complejos. Sin embargo, una desventaja es que el Fletcher Checksum solo puede detectar errores, no corregirlos. Esto limita su utilidad en aplicaciones donde la corrección de errores es importante. Además, aunque es eficaz en la mayoría de los casos, hay ciertos patrones de errores múltiples que podrían no ser detectados.