

Esquemas de detección y corrección de errores

El laboratorio se dividió en dos partes. La primera consistió en implementar algoritmos de detección y corrección, con el propósito de analizar el funcionamiento de cada uno, así como identificar las ventajas y desventajas de cada uno. Ahora bien, la segunda parte consistió en desarrollar una aplicación que implementa los algoritmos mencionados anteriormente para la transmisión y recepción de mensajes, en base a una arquitectura de capas con distintos servicios.

Parte 1: Implementación de Algoritmos

Para el siguiente ejercicio, escogimos un algoritmo de corrección de errores (Código de Hamming) y dos de detección de errores (Fletcher checksum y CRC-32). El objetivo era investigar y analizar cada algoritmo para determinar los mejores casos de uso para cada uno. Además, se decidió realizar el código para el emisor en JavaScript y para el receptor en Python.

Los mensajes originales fueron los siguientes:

- Mensaje 1: 10110100
- Mensaje 2: 010001011010
- Mensaje 3: 1101001110100110

Código de Hamming

Este algoritmo permite detectar y corregir errores introducidos en los bits de un mensaje que se transmiten de un lado a otro a través de un medio de comunicación. Los bits del mensaje se codifican mediante el uso de bits de paridad que se ubican en posiciones específicas (posiciones de múltiplos de 2). En el lado del receptor, se recalculan estos bits, de manera que se puedan ubicar los bits de error y aplicar la corrección necesaria.

Implementación

Los bits de paridad se muestran marcados en verde y los bits que se cambiaron para probar la detección y corrección de errores están en rojo.

Emisor:

Mensajes codificados:

- Mensaje 1: 001001110100

```
m=8, r=4, n=12 (condición:  $m+r+1 \leq 2^r$ )
Posiciones: 1 2 3 4 5 6 7 8 9 10 11 12
Mapa inicial (P=paridad, número=dato):
  P P 1 P 0 1 1 P 0 1 0 0
P1 cubre posiciones: 1, 3, 5, 7, 9, 11 → XOR = 0
P2 cubre posiciones: 2, 3, 6, 7, 10, 11 → XOR = 0
P4 cubre posiciones: 4, 5, 6, 7, 12 → XOR = 0
P8 cubre posiciones: 8, 9, 10, 11, 12 → XOR = 1
Trama codificada final: 001001110100
```

- Mensaje 2: 10001000010110100

```
m=12, r=5, n=17 (condición:  $m+r+1 \leq 2^r$ )
Posiciones: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Mapa inicial (P=paridad, número=dato):
  P P 0 P 1 0 0 P 0 1 0 1 1 0 1 P 0
P1 cubre posiciones: 1, 3, 5, 7, 9, 11, 13, 15, 17 → XOR = 1
P2 cubre posiciones: 2, 3, 6, 7, 10, 11, 14, 15 → XOR = 0
P4 cubre posiciones: 4, 5, 6, 7, 12, 13, 14, 15 → XOR = 0
P8 cubre posiciones: 8, 9, 10, 11, 12, 13, 14, 15 → XOR = 0
P16 cubre posiciones: 16, 17 → XOR = 0
Trama codificada final: 10001000010110100
```

- Mensaje 3: 111010100011101000110

```
m=16, r=5, n=21 (condición: m+r+1 ≤ 2^r)
Posiciones: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Mapa inicial (P=paridad, número=dato):
  P P 1 P 1 0 1 P 0 0 1 1 1 0 1 P 0 0 1 1 0
P1 cubre posiciones: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 → XOR = 1
P2 cubre posiciones: 2, 3, 6, 7, 10, 11, 14, 15, 18, 19 → XOR = 1
P4 cubre posiciones: 4, 5, 6, 7, 12, 13, 14, 15, 20, 21 → XOR = 0
P8 cubre posiciones: 8, 9, 10, 11, 12, 13, 14, 15 → XOR = 0
P16 cubre posiciones: 16, 17, 18, 19, 20, 21 → XOR = 0
Trama codificada final: 111010100011101000110
```

Receptor:

Mensaje 1:

- Mensaje OK: 001001110100

```
n=12, r=4
Trama recibida: 001001110100
Síndrome (dec)=0, (bin)=0000
msg1_ok.txt -> OK 10110100
```

- Mensaje con un error: 001001100100

```
n=12, r=4
Trama recibida: 001001100100
Síndrome (dec)=8, (bin)=1000
Se corrige bit en posición 8.
Trama corregida: 001001110100
msg1 err1.txt -> FIX pos=8 10110100
```

- Mensaje con dos o más errores: 111001110100

```
n=12, r=4
Trama recibida: 111001110100
Síndrome (dec)=3, (bin)=0011
Se corrige bit en posición 3.
Trama corregida: 110001110100
msg1_err2.txt -> FIX pos=3 00110100
```

Como se puede evidenciar, los bits que se cambiaron son de paridad (posiciones 1 y 2), pero el algoritmo muestra un único error en la posición del mensaje, haciendo que se corrija el bit equivocado y devolviendo el mensaje equivocado.

Mensaje 2:

- Mensaje OK: 10001000010110100

```
n=17, r=5
Trama recibida: 10001000010110100
Síndrome (dec)=0, (bin)=00000
msg2_ok.txt -> OK 010001011010
```

- Mensaje con un error: 10001000010110110

```
n=17, r=5
Trama recibida: 10001000010110110
Síndrome (dec)=16, (bin)=10000
Se corrige bit en posición 16.
Trama corregida: 10001000010110100
msg2_err1.txt -> FIX pos=16 010001011010
```

- Mensaje con dos o más errores: 01001000010110110

```
n=17, r=5
Trama recibida: 01001000010110110
Síndrome (dec)=19, (bin)=10011
msg2_err2.txt -> DROP (errores no corregibles)
```

En este caso, al sumar las posiciones de donde se encuentran los errores, no puede detectar correctamente el/los errores, puesto que identifica que el bit erróneo está en la posición 19, cuando la longitud del mensaje codificado es de 17 bits.

Mensaje 3:

- Mensaje OK: 111010100011101000110

```
n=21, r=5
Trama recibida: 111010100011101000110
Síndrome (dec)=0, (bin)=00000
msg3_ok.txt -> OK 1101001110100110
```

- Mensaje con un error: 111010110011101000110

```
n=21, r=5
Trama recibida: 111010110011101000110
Síndrome (dec)=8, (bin)=01000
Se corrige bit en posición 8.
Trama corregida: 111010100011101000110
msg3_err1.txt -> FIX pos=8 1101001110100110
```

- Mensaje con dos errores o más: 011110110011101000110

```
n=21, r=5
Trama recibida: 011110110011101000110
Síndrome (dec)=13, (bin)=01101
Se corrige bit en posición 13.
Trama corregida: 011110110011001000110
msg3_err2.txt -> FIX pos=13 1101001100100110
```

Como se observa en la imagen anterior, los bits que se cambiaron son de paridad (posiciones 1, 4 y 8), pero el algoritmo muestra un único error en la posición del mensaje, haciendo que se corrija el bit equivocado y devolviendo el mensaje equivocado.

En conclusión, el código de Hamming está diseñado para detectar y corregir únicamente un bit erróneo. Tiene una capacidad limitada para manejar múltiples errores.

¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error?

- En este caso, se pueden manipular los bits para que el algoritmo no sea capaz de detectar el error. Por ejemplo, en el caso de dos o más errores en el mensaje 2, se observa como la posición calculada del bit de error se encuentra fuera del largo de la cadena, por lo que este no es capaz de detectar un error en el mensaje y mucho menos corregirlo, por lo que fácilmente podría detectarse como un falso positivo en una implementación real.

CRC-32

El CRC-32, es un algoritmo de suma de verificación de redundancia cíclica que como dice su nombre, genera un valor de 32 bits a partir de una secuencia de datos. Este algoritmo es utilizado para detectar cambios entre los datos de origen y destino, es utilizado en formatos de archivos como Gzip y PNG.

El algoritmo se basa en un polinomio de grado 32, en el caso de este laboratorio se utilizó el polinomio generador IEEE 802.3, representado por 0xEDB88320 en hexadecimal.

Una vez definido el polinomio se siguen los siguientes pasos:

1. Se toma el mensaje original $M(x)$ en bits.
2. Se agregan 32 ceros al final (espacio para el CRC).
3. Se divide el mensaje extendido entre el polinomio generador, usando división binaria (XOR).
4. El residuo de 32 bits es el CRC calculado, se concatena al final de la cadena en binario original y se envía.

Ahora para la decodificación se hace lo siguiente:

1. Se recibe el mensaje enviado por el emisor
2. Se divide toda la trama entre el polinomio generador.
 - Si el residuo = 0 entonces la trama es válida.
 - Si el residuo $\neq 0$ entonces la trama está corrupta.

Implementación

Los mensajes codificados que recibe el **Decoder.py** tienen el formato de Mensaje Original + CRC-32, por lo que tendrán una longitud de 32 bits adicionales a la longitud original.

Emisor:

Mensajes codificados:

1. Mensaje 1: 001001110100
 - o CRC-32: 000011101101110000001110000100001010

```
=== Procesando tests/msg1.txt ===
Se agregó padding de 4 bits para completar bytes de 8 bits
✓ tests/msg1.txt → C:\Users\drkfa\U\REDES\LAB2-REDES\Parte1\CRC-32\out\msg1_crc32.txt
Mensaje original: 12 bits
Mensaje con CRC32: 48 bits
Overhead: 36 bits (32 bits de CRC + padding)
CRC32: 11101101110000001110000100001010
```

2. Mensaje 2: 010001011010

- CRC-32: 0000000110100000001000010100101011

```
=== Procesando tests/msg2.txt ===  
Se agregó padding de 4 bits para completar bytes de 8 bits  
✓ tests/msg2.txt → C:\Users\drkfa\U\REDES\LAB2-REDES\Parte1\CRC-32\out\msg2_crc32.txt  
Mensaje original: 12 bits  
Mensaje con CRC32: 48 bits  
Overhead: 36 bits (32 bits de CRC + padding)  
CRC32: 0001101000000001000010100101011
```

3. Mensaje 3: 1101001110100110

- CRC-32: 1101010001111001100000101111110

```
=== Procesando tests/msg3.txt ===  
✓ tests/msg3.txt → C:\Users\drkfa\U\REDES\LAB2-REDES\Parte1\CRC-32\out\msg3_crc32.txt  
Mensaje original: 16 bits  
Mensaje con CRC32: 48 bits  
Overhead: 32 bits (32 bits de CRC + padding)  
CRC32: 1101010001111001100000101111110
```

Receptor:

- El mensaje original estará en **negrita**, el CRC calculado subrayado de amarillo y los bits modificados serán rojos.

1. Mensaje 1:

- Mensaje OK:

001001110100000011101101110000001110000100001010

```
-----  
msg1_crc32_ok.txt -> OK 0010011101000000  
-----
```


- Mensaje con un error:

001001110100000011101101111000001110000100001010

```
-----  
msg1_crc32_err1.txt -> ERROR - Se detectaron errores  
El mensaje se descarta por detectar errores.  
-----
```

- Mensaje con dos o más errores:

0010011101001101110011010101000011100000000001010

```
-----  
msg1_crc32_err2.txt -> ERROR - Se detectaron errores  
El mensaje se descarta por detectar errores.  
-----
```

2. Mensaje 2:

- Mensaje OK:

01000101101000000011010000000010000101001010111

```
-----  
msg2_crc32_ok.txt -> OK 0100010110100000  
-----
```

- Mensaje con un error:

010001011011000000011010000000010000101001010111

```
-----  
msg2_crc32_err1.txt -> ERROR - Se detectaron errores  
El mensaje se descarta por detectar errores.  
-----
```

- Mensaje con dos o más errores:

01000101101000000011001000000010000001001010111

```
-----  
msg2_crc32_err2.txt -> ERROR - Se detectaron errores  
El mensaje se descarta por detectar errores.  
-----
```

3. Mensaje 3:

- Mensaje OK:

110100111010011011010100011110011000001011111110

```
-----  
msg3_crc32_ok.txt -> OK 1101001110100110  
-----
```

- Mensaje con un error:

110100111010011011010100011110010000001011111110

```
-----  
msg3_crc32_err1.txt -> ERROR - Se detectaron errores  
El mensaje se descarta por detectar errores.  
-----
```

- Mensaje con dos errores o más:

110100111010011001010101100111100011100010011111110

```
-----
msg3_crc32_err2.txt -> ERROR - Se detectaron errores
El mensaje se descarta por detectar errores.
```

Todos los errores generados en las pruebas fueron detectados correctamente por el algoritmo CRC-32, gracias a su capacidad para identificar discrepancias en los datos mediante el cálculo del checksum basado en el polinomio generador. Sin embargo, al investigar sobre las limitaciones del **CRC-32**, se descubrió que es posible encontrar colisiones, es decir, distintos mensajes que producen el mismo valor de **CRC**. En tales casos, una manipulación intencional de los datos podría pasar desapercibida, ya que el receptor aceptaría el mensaje como válido.

¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error?

Si es posible, a continuación una demostración y una breve explicación de porqué es posible.

- **MensajeCollision1:**

- **Original:**

011100000110110001110101011011010110001100101011100
1101110011

- **CRC-32:** 01001101110110110000110000100101

```
=== Procesando tests/msg_collision1.txt ===
✓ tests/msg_collision1.txt -> C:\Users\drkfa\U\REDES\LAB2-REDES\Parte1\CRC-32\out\msg_collision1_crc32.txt
Mensaje original: 64 bits
Mensaje con CRC32: 96 bits
Overhead: 32 bits (32 bits de CRC + padding)
CRC32: 01001101110110110000110000100101
```

- **MensajeCollision2:**

- **Original:**

011000100111010101100011011010110110010101110010011011
1101101111

- **CRC-32:** 01001101110110110000110000100101

```
=== Procesando tests/msg_collision2.txt ===
✓ tests/msg_collision2.txt → C:\Users\drkfa\U\REDES\LAB2-REDES\Parte1\CRC-32\out\msg_collision2_crc32.txt
Mensaje original: 64 bits
Mensaje con CRC32: 96 bits
Overhead: 32 bits (32 bits de CRC + padding)
CRC32: 01001101110110110000110000100101
```

Receptor:

```
-----
msg_collision1.txt → OK 01110000011011000111010101101101101100011001010111001101110011
```

```
-----
msg_collision2.txt → OK 0110001001110101011000110110101101100101011100100110111101101111
```

El **CRC-32**, al ser un hash de 32 bits, tiene un espacio finito de valores (2^{32}). Por el principio del palomar, múltiples mensajes pueden mapearse al mismo **CRC**. En este caso, los dos mensajes generan el mismo CRC, por lo que una manipulación que cambie un mensaje por otro con el mismo CRC no será detectada, ya que el receptor valida el CRC sin distinguir los datos originales. Esta vulnerabilidad tiene una probabilidad muy baja para errores aleatorios (aproximadamente de 1 en 4.3 billones), sin embargo es posible intencionalmente.

Fletcher Checksum

Fletcher Checksum es un algoritmo de verificación que permite detectar errores en la transmisión o almacenamiento de datos. Funciona dividiendo el mensaje en bloques de tamaño fijo (por ejemplo, 4, 8, 16 o 32 bits) y calculando dos sumas acumulativas (sum1 y sum2) sobre esos bloques, usando aritmética modular.

Implementación

En el caso de la codificación, el mensaje original se divide en bloques del tamaño especificado (por ejemplo, 4, 8, 16 o 32 bits) donde se calcula el checksum de Fletcher sumando los bloques de forma acumulativa (sum1 y sum2), usando un módulo específico para el tamaño de bloque. El resultado del checksum (sum1 y sum2) se convierte a binario y se concatena al final del mensaje original (con padding si es necesario).

El archivo de salida contiene:

Mensaje codificado = Mensaje original (con padding si aplica) + Checksum (sum1 + sum2)

Ahora bien, en el caso de la decodificación se separa el mensaje recibido en dos partes: los datos originales y el checksum recibido (los últimos $2 \times \text{block_size}$ bits). En donde calcula el checksum de los datos recibidos y lo compara con el checksum recibido. Si ambos coinciden, el mensaje es válido (OK); si no, se detecta un error.

Emisor:

Mensajes codificados:

- **Mensaje 1:**

- Fletcher 8 | Mensaje: 10101100
- Checksum = **00001011**
- Mensaje completo: **1011010000001011**
(datos + sum1 + sum2)

```
PS C:\Users\rebe1\OneDrive\Documentos\GitHub\LAB2-REDES
-size=4 --verbose

=== Procesando in/fletcher8.txt ===
Mensaje original: 10110100 (8 bits)

--- Procedimiento detallado Fletcher4 ---
No se agregó padding.
Bloques:
  Bloque 1: 1011 (decimal: 11)
  Bloque 2: 0100 (decimal: 4)

Cálculo de sumas parciales (modulo 15):
  Iteración 1: sum1=11 (1011), sum2=11 (1011)
  Iteración 2: sum1=0 (0000), sum2=11 (1011)

Checksum:
  sum1: 0 (0000)
  sum2: 11 (1011)
  Checksum combinado: 00001011

Mensaje final:
  Datos (con padding): 10110100
  Checksum:           00001011
  Mensaje completo:   1011010000001011
  [datos][checksum]
✓ Archivo generado: C:\Users\rebe1\OneDrive\Documentos\
txt
  Mensaje original: 8 bits
  Mensaje con Fletcher: 16 bits
  Overhead: 8 bits (8 bits de checksum + padding)
```

- **Mensaje 2:**

- Fletcher 16 | Mensaje: 010001011010
- Checksum = **1110010100101011**
- Mensaje completo: **01000101101000001110010100101011**
(datos + sum1 + sum2)

```

PS C:\Users\rebe1\OneDrive\Documentos\GitHub\LAB2-REDES\Parte1\FletcherChecksum> k-size=8 --verbose

=== Procesando in/fletcher16.txt ===
Mensaje original: 010001011010 (12 bits)

--- Procedimiento detallado Fletcher8 ---
Se agregó padding de 4 bits para completar bloques de 8 bits.
Bloques:
  Bloque 1: 01000101 (decimal: 69)
  Bloque 2: 10100000 (decimal: 160)

Cálculo de sumas parciales (modulo 255):
  Iteración 1: sum1=69 (01000101), sum2=69 (01000101)
  Iteración 2: sum1=229 (11100101), sum2=43 (00101011)

Checksum:
  sum1: 229 (11100101)
  sum2: 43 (00101011)
  Checksum combinado: 1110010100101011

Mensaje final:
  Datos (con padding): 0100010110100000
  Checksum:           1110010100101011
  Mensaje completo:   01000101101000001110010100101011
  [datos][checksum]
✓ Archivo generado: C:\Users\rebe1\OneDrive\Documentos\GitHub\LAB2-REDES\Parte1\FletcherChecksum\in/fletcher16.txt
Mensaje original: 12 bits
Mensaje con Fletcher: 32 bits
Overhead: 20 bits (16 bits de checksum + padding)

```

● Mensaje 3:

- Fletcher 32 | Mensaje: 1101001110100110
- Checksum = **11010011101001101101001110100110**
- Mensaje completo:

110100111010011011010011101001101101001110100110
 (datos + sum1 + sum2)

```

PS C:\Users\rebe1\OneDrive\Documentos\GitHub\LAB2-REDES\Parte1\FletcherChecksum> node k-size=16 --verbose

=== Procesando in/fletcher32.txt ===
Mensaje original: 1101001110100110 (16 bits)

--- Procedimiento detallado Fletcher16 ---
No se agregó padding.
Bloques:
  Bloque 1: 1101001110100110 (decimal: 54182)

Cálculo de sumas parciales (modulo 65535):
  Iteración 1: sum1=54182 (1101001110100110), sum2=54182 (1101001110100110)

Checksum:
  sum1: 54182 (1101001110100110)
  sum2: 54182 (1101001110100110)
  Checksum combinado: 11010011101001101101001110100110

Mensaje final:
  Datos (con padding): 1101001110100110
  Checksum:           11010011101001101101001110100110
  Mensaje completo:   110100111010011011010011101001101101001110100110
  [datos][checksum]
✓ Archivo generado: C:\Users\rebe1\OneDrive\Documentos\GitHub\LAB2-REDES\Parte1\FletcherChecksum\in/fletcher32.txt
Mensaje original: 16 bits
Mensaje con Fletcher: 48 bits
Overhead: 32 bits (32 bits de checksum + padding)

```

Receptor:

Mensaje 1:

- Mensaje OK:

```
fletcher8_fletcher4.txt -> OK 10110100  
Checksum = 10110000 = 10110000
```

- Mensaje con un error:

1001010000001011

```
fletcher8_fletcher4.txt -> ERROR - Se detectaron errores  
El mensaje se descarta por detectar errores.  
Datos recibidos (sin checksum): 10010100  
Dato (bloque 4 bits)  Sum1  Sum2  
1001 ( 9)  1010 ( 10)  1011 ( 11)  
0100 ( 4)  1110 ( 14)  1010 ( 10)  
Checksum = 01111101 ≠ 10110000. ERROR
```

- Mensaje con dos o más errores:

1011011000001111

```
fletcher8_fletcher4.txt -> ERROR - Se detectaron errores  
El mensaje se descarta por detectar errores.  
Datos recibidos (sin checksum): 10110110  
Dato (bloque 4 bits)  Sum1  Sum2  
1011 ( 11)  1100 ( 12)  1101 ( 13)  
0110 ( 6)  0011 ( 3)  0001 ( 1)  
Checksum = 11010010 ≠ 11110000. ERROR
```

Mensaje 2:

- Mensaje OK:

```
fletcher16_fletcher8.txt -> OK 0100010110100000  
Checksum = 0010101111100101 = 0010101111100101
```

- Mensaje con un error:

01001101101000001110010100101011


```
fletcher16_fletcher8.txt -> ERROR - Se detectaron errores
El mensaje se descarta por detectar errores.
Datos recibidos (sin checksum): 0100110110100000
Dato (bloque 8 bits)  Sum1  Sum2
01001101 ( 77)  01001110 ( 78)  01001111 ( 79)
10100000 (160)  11101110 (238)  00111110 ( 62)
Checksum = 0011101111101101 ≠ 0010101111100101. ERROR
```

- Mensaje con dos o más errores:
01001101101100011110010100101010

```
fletcher16_fletcher8.txt -> ERROR - Se detectaron errores
El mensaje se descarta por detectar errores.
Datos recibidos (sin checksum): 0100110110110001
Dato (bloque 8 bits)  Sum1  Sum2
01001101 ( 77)  01001110 ( 78)  01001111 ( 79)
10110001 (177)  00000000 ( 0)  01001111 ( 79)
Checksum = 0100110011111110 ≠ 0010101111100101. ERROR
```

Mensaje 3:

- Mensaje OK:

```
fletcher32_fletcher16.txt -> OK 1101001110100110
Checksum = 11010011101001101101001110100110 = 11010011101001101101001110100110
```

- Mensaje con un error:
100100111010011001010011101001100101001110100110

```
fletcher32_fletcher16.txt -> ERROR - Se detectaron errores
El mensaje se descarta por detectar errores.
Datos recibidos (sin checksum): 1001001110100110
Dato (bloque 16 bits)  Sum1  Sum2
1001001110100110 (37798)  1001001110100111 (37799)  1001001110101000 (37800)
Checksum = 10010011101001101001001110100110 ≠ 01010011101001100101001110100110. ERROR
```

- Mensaje con dos errores o más:
100010111010011011010011101001101101001110100110

```
fletcher32_fletcher16.txt -> ERROR - Se detectaron errores
El mensaje se descarta por detectar errores.
Datos recibidos (sin checksum): 1000101110100110
Dato (bloque 16 bits)  Sum1  Sum2
1000101110100110 (35750)  1000101110100111 (35751)  1000101110101000 (35752)
Checksum = 10001011101001101000101110100110 ≠ 11010011101001101101001110100110. ERROR
```

Todos los errores generados en las pruebas fueron detectados correctamente por el algoritmo Fletcher Checksum, gracias a sus sumas parciales sum1 y sum2. Sin embargo, existen limitaciones como puede ser el que distintos mensajes pueden producir el mismo checksum (colisiones), por lo que en casos de manipulación intencional, el receptor podría aceptar datos incorrectos como válidos. Por eso, para

mayor seguridad se recomienda complementar Fletcher con otros métodos de verificación.

¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error?

Sí, es posible manipular los bits de tal manera que el algoritmo de Fletcher checksum no detecte el error. Esto se debe a que, aunque el método es bueno para encontrar fallos simples como cuando un solo bit cambia, sigue siendo propenso a fallos frente a ciertos patrones de error más complejos. Por ejemplo, si se modifican dos bytes de forma que el incremento en uno se compensa con la disminución en otro, la suma total y la suma acumulativa que usa Fletcher pueden quedar iguales, haciendo que el error pase desapercibido.

Comparación

Ventajas y Desventajas:

Hamming Code:

- **Ventajas:**
 - Fácil de entender e implementar
 - Altamente eficiente para la detección y corrección de errores de un solo bit.
- **Desventajas:**
 - Solo corrige un error de un solo bit. En caso de haber múltiples errores, su capacidad de corrección más que detección es limitada.
 - En términos de eficiencia, es más lento a comparación de los algoritmos de detección, porque necesita realizar validaciones extra.
 - Es sensible al ruido.

CRC-32:

- **Ventajas:**
 - CRC-32 detecta con alta probabilidad errores simples, dobles, de ráfaga y permutaciones.
 - Es bastante simple para implementar.
 - Es confiable para la integridad de datos.
- **Desventajas:**
 - No está diseñado para proteger mensajes, únicamente tiene como función verificar que el mensaje recibido es correcto.
 - Este algoritmo es más costoso que Hamming o Checksum.

- Requiere que el decodificador conozca el polinomio estándar utilizado para la codificación.
- Como se observó en los ejemplos, hay casos muy específicos donde dos cadenas tienen el mismo CRC-32, por lo que con una manipulación detallada es posible que se dé como válido un mensaje que no lo es.

Fletcher Checksum:

- **Ventajas:**
 - Fletcher Checksum puede detectar no solo errores de un solo bit, sino también muchos errores de transposición y de múltiples bits, debido a su doble suma acumulativa (sum1 y sum2).
 - El algoritmo sólo requiere operaciones de suma y módulo, sin necesidad de multiplicaciones y divisiones complejas. Esto permite implementarlo fácilmente en hardware o software, y procesar grandes volúmenes de datos.
- **Desventajas:**
 - No garantiza la detección de todos los patrones de error, especialmente si ocurren errores correlacionados en posiciones específicas. Por ejemplo, ciertos errores múltiples pueden anularse entre sí y pasar desapercibidos.
 - No está diseñado para proteger contra ataques intencionados ni para verificar la autenticidad de los datos. Es vulnerable a manipulaciones deliberadas.

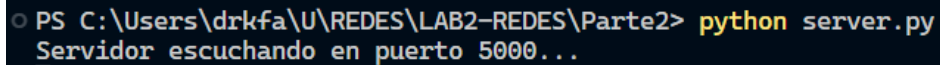
Parte 2: Conexión y Testing

Para la segunda parte del laboratorio, se implementó un cliente (Emisor) y un servidor (Receptor) para el paso de mensajes con los 3 algoritmos implementados. El cliente se realizó con JavaScript y el servidor con Python, la conexión se implementó con Sockets mediante TCP. Adicionalmente se simuló el ruido (como representación de interferencia de parte del Emisor), para determinar cuál bit se volteaba, nos basamos en una probabilidad definida previamente.

Para la ejecución de los programas, se debe de ejecutar en consolas separadas lo siguiente:

1. Servidor:

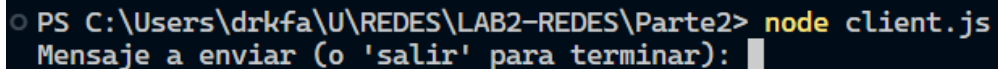
`python server.py`



```
PS C:\Users\drkfa\U\REDES\LAB2-REDES\Parte2> python server.py
Servidor escuchando en puerto 5000...
```

2. Cliente:

`node client.js`



```
PS C:\Users\drkfa\U\REDES\LAB2-REDES\Parte2> node client.js
Mensaje a enviar (o 'salir' para terminar):
```

Esto iniciará un servidor esperando por mensajes y un cliente el cuál le solicitará al usuario que ingrese el mensaje que desea enviar, junto al algoritmo con el que se va a enviar.

Como ejemplo se utilizó el algoritmo de Hamming, primero se ejecutó con una probabilidad de ruido de 0.001, por lo que muy difícilmente se aplicaría algún tipo de ruido.

```
PS C:\Users\drkfa\U\REDES\LAB2-REDES\Parte2> node client.js
Mensaje a enviar (o 'salir' para terminar): Hola
Algoritmo (Hamming/Fletcher/CRC): Hamming
Convirtiendo mensaje a binario:
Hola → 01001000011011110110001100001
Enviado: 11001001100001110111101101100010100001
--- Listo para enviar otro mensaje ---

PS C:\Users\drkfa\U\REDES\LAB2-REDES\Parte2> python server.py
Servidor escuchando en puerto 5000...
{'algo': 'hamming', 'trama': '11001001100001110111101101100010100001'}
=====
Trama recibida: 11001001100001110111101101100010100001
[hamming] Status: OK
Mensaje recibido: Hola
█
```

Ahora con una probabilidad de 0.005, donde es más seguro que haya como mínimo 1 error:

```
--- Listo para enviar otro mensaje ---
Mensaje a enviar (o 'salir' para terminar): Hola
Algoritmo (Hamming/Fletcher/CRC): hamming
Convirtiendo mensaje a binario:
Hola → 01001000011011110110001100001
Enviado: 10001001100001110111101101100010100001

{'algo': 'hamming', 'trama': '10001001100001110111101101100010100001'}
=====
Trama recibida: 10001001100001110111101101100010100001
[hamming] Status: FIX
Corrección Hamming: pos=2
Mensaje recibido: Hola
{'algo': 'crc', 'trama': '011000001110100010110111101111001000011'}
=====
```

Podemos ver que Hamming logró encontrar el bit flippeado y corrigió la trama, desplegando así el mensaje original “Hola”.

Para una última ejecución, se aumentó el ruido a 0.01 con el fin de que se volteen 2 o más bits:

```
PS C:\Users\drkfa\U\REDES\LAB2-REDES\Parte2> node client.js
Mensaje a enviar (o 'salir' para terminar): Hola a todos
Algoritmo (Hamming/Fletcher/CRC): Hamming
Convirtiendo mensaje a binario:
Hola a todos → 01001000011011110110001100001001000000110000100100
000011010001101111011001000110111101110011
Enviado: 01011001100001100111101101100010100101001000000110000100100
00001101000100111011001000110111101110011
--- Listo para enviar otro mensaje ---
Mensaje a enviar (o 'salir' para terminar): █

PS C:\Users\drkfa\U\REDES\LAB2-REDES\Parte2> python server.py
Servidor escuchando en puerto 5000...
{'algo': 'hamming', 'trama': '0101100110000110011110110110001010010100100100000011001000100111011001100111'}
=====
Trama recibida: 0101100110000110011110110110001010010100100100000011000
0100100000001110100010011110110010001101110110011
[hamming] Status: DROP
Mensaje recibido: Hole a tOdos
█
```

En este caso podemos observar que se hizo DROP del mensaje debido a que Hamming no pudo corregir todos los errores, desplegando luego el mensaje erróneo donde en lugar de “Hola a todos”, dice “Hole a tOdos”.

Testing

Para comparar la eficiencia de los algoritmos, se implementó una función de prueba que opera tanto en el cliente como en el servidor. Al inicio de cada prueba, se genera automáticamente una carpeta con un **timestamp** único para almacenar los datos de la ejecución.

Proceso del cliente y servidor

El cliente primero escribe un reporte detallado (client_report.csv) que incluye:

- El algoritmo de codificación utilizado.
- El mensaje original en formato ASCII y binario.
- La longitud de la cadena en ambos formatos.
- El mensaje codificado.
- La probabilidad de ruido del canal.

- El número de bits invertidos.

Posteriormente, **el servidor**, al recibir cada mensaje, utiliza un identificador único para leer los datos enviados por el cliente y realiza una comparación. Luego, genera su propio reporte (server_report.csv) con la siguiente información:

- ID del mensaje.
- Algoritmo utilizado.
- El mensaje recibido.
- Un indicador de éxito (si se recibió correctamente o si fue necesaria una corrección de trama).

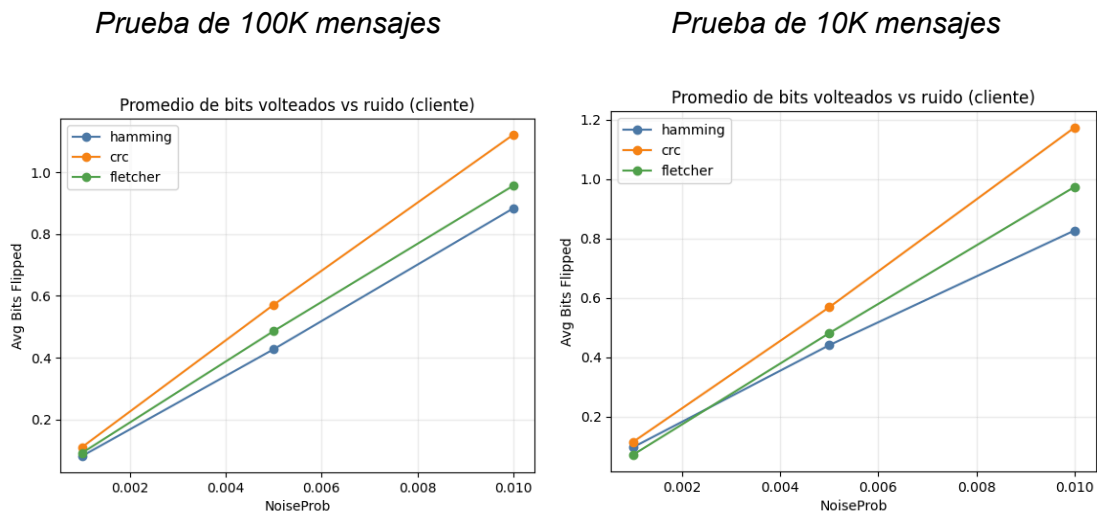
Para garantizar la precisión, el servidor también realiza una verificación adicional. Compara los mensajes marcados como exitosos con el mensaje original para detectar "falsos positivos". Si el mensaje no coincide, registra el mensaje real y el erróneo en un archivo (errors.csv).

Para evaluar el rendimiento de los algoritmos, se diseñó un proceso de prueba que garantiza una comparación equitativa.

1. **Generación de Mensajes:** En cada iteración se genera un mensaje aleatorio único que luego se envía al algoritmo y probabilidad de error correspondientes.
2. **Distribución de Pruebas:** Los mensajes se envían de forma equitativa a los tres algoritmos, y dentro de cada algoritmo, se prueban con tres probabilidades de error distintas: **0.001**, **0.005** y **0.01**. Esto asegura que todos los escenarios posibles reciban la misma cantidad de mensajes, a pesar de que cada uno sea único.
3. **Análisis:** El análisis se llevó a cabo con dos pruebas principales: una con **10,000** mensajes y otra con **100,000** mensajes, lo que permite observar el comportamiento de los algoritmos a diferentes escalas.
4. **Automatización de reportes:** Al finalizar el test, se generan varias gráficas describiendo los datos capturados junto a un archivo report.md donde se despliegan las gráficas generadas.

Resultados

Figura 1. Gráfico de promedio de bits volteados vs ruido

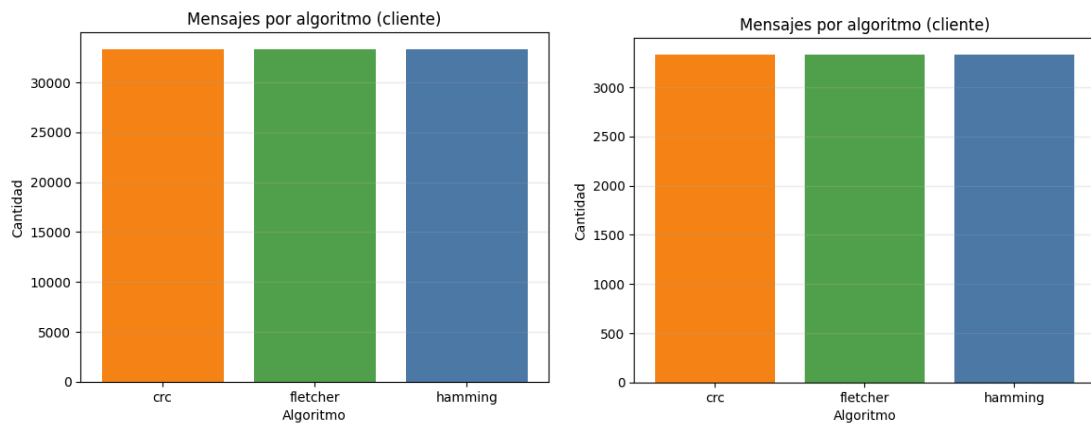


En la gráfica se observa la relación entre la probabilidad de ruido en el canal (NoiseProb) y el promedio de bits volteados (Avg Bits Flipped) para los tres algoritmos seleccionados. En la prueba de los 100k y los 10k se observa que a medida que aumenta la probabilidad de ruido, el promedio de bits volteados por mensaje también incrementa de manera casi lineal para los tres algoritmos. En ambos se observa que CRC tiende a presentar un promedio ligeramente mayor de bits volteados en comparación con Hamming y Fletcher.

Figura 2. Gráfico de mensajes por algoritmo

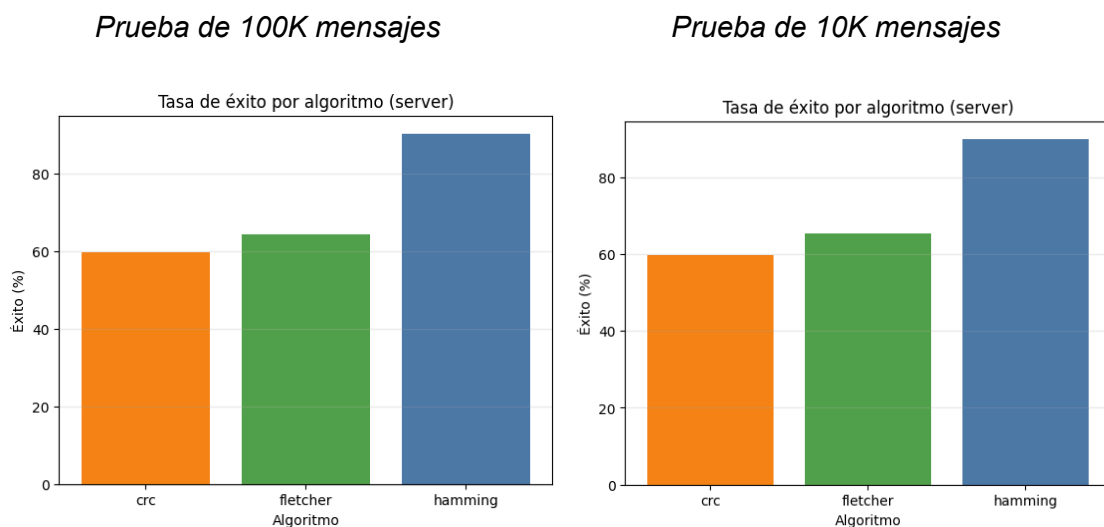
Prueba de 100K mensajes

Prueba de 10K mensajes



En las gráficas se observa la cantidad de mensajes enviados por el cliente utilizando cada uno de los algoritmos. Se observa que la distribución de mensajes es prácticamente equitativa para los tres algoritmos, con alrededor de 33,333 / 3,333 mensajes enviados por cada uno.

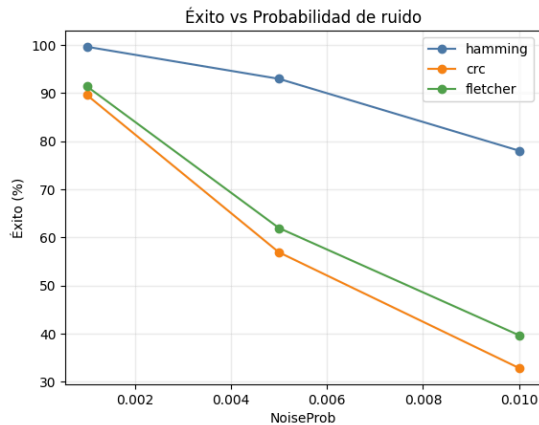
Figura 3. Gráfico de tasa de éxito por algoritmo



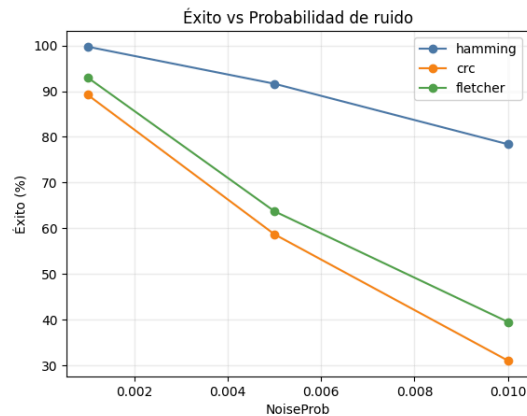
En los gráficos se observa el porcentaje de éxito alcanzado por el servidor al recibir mensajes procesados con cada uno de los algoritmos. Se observa que el algoritmo de Hamming alcanza la mayor tasa de éxito, superando el 90%. Esto se debe a su capacidad de corregir errores de un solo bit, permitiendo recuperar mensajes que de otro modo estos serían descartados. En el caso Fletcher muestra una tasa de éxito intermedia, alrededor del 65%, mientras que CRC tiene la tasa más baja, cercana al 60%. Esto refleja que, aunque CRC y Fletcher son muy efectivos para detectar errores, no pueden corregirlos.

Figura 4. Gráfico de Éxito vs Probabilidad de Ruido

Prueba de 100K mensajes

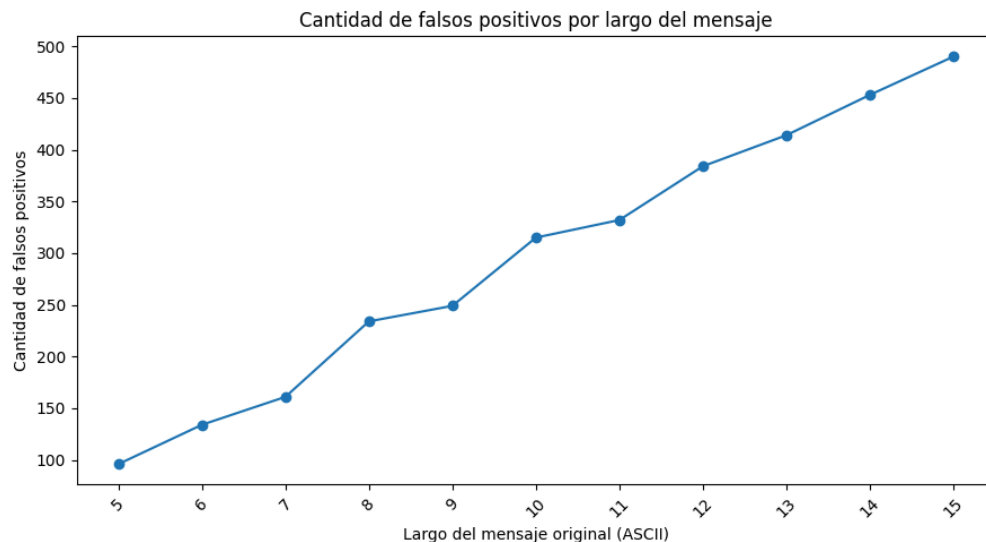


Prueba de 10K mensajes



En estos gráficos se observa cómo varía el porcentaje de éxito en la recepción de mensajes según la probabilidad de ruido en el canal, para los algoritmos. Hamming mantiene la mayor tasa de probabilidad, en el caso Fletcher y CRC presentan una caída más pronunciada, ya que solo detectan errores y no pueden corregirlos, por lo que cualquier alteración en el mensaje reduce la tasa de éxito.

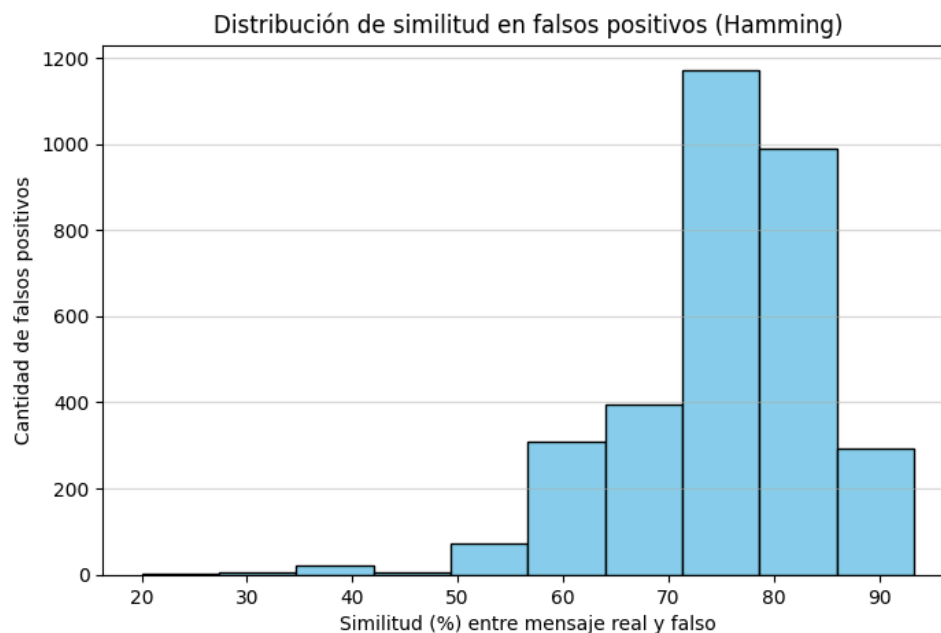
Figura 5. Cantidad de falsos positivos vs largo de mensaje



En la gráfica se observa que, a medida que aumenta la longitud de los mensajes enviados, también crece la probabilidad de que ocurran falsos positivos. Esto sucede en el caso del algoritmo de Hamming porque, cuanto mayor es la cadena original, se generan más bits de paridad. Al incrementarse el número total de bits, aumenta la probabilidad de que ocurran múltiples errores simultáneos. Dichos errores, en ciertos

patrones, pueden “anularse” entre sí en el cálculo de paridades, provocando que el receptor acepte como válido un mensaje que en realidad está corrupto.

Figura 6. Distribución de similitud de falsos positivos



En la gráfica de distribución se aprecia que la mayoría de los falsos positivos presentan una similitud de alrededor del 80% con el mensaje original, lo que indica que los errores suelen ser parciales y no alteran por completo el contenido. Sin embargo, se observa que sólo un número reducido de falsos positivos cae por debajo del 50% de similitud, lo que corresponde a los casos más graves, en los que el mensaje recibido difiere de manera significativa del original.

Discusión

En este laboratorio se evaluaron tres algoritmos de detección y corrección de errores: Hamming, CRC-32 y Fletcher Checksum, tanto en simulaciones de transmisión de mensajes como en un entorno cliente-servidor real. Se realizaron pruebas masivas (10K y 100K mensajes) para analizar el comportamiento de cada algoritmo bajo diferentes niveles de ruido.

En los gráficos de arriba, como se muestra en la Figura 2. se evidencia que la cantidad de mensajes enviados por cada algoritmo el cuál fue prácticamente igual, garantizando una comparación balanceada de los resultados. Las gráficas de la Figura 1. muestran que a medida que aumenta la probabilidad de ruido, el promedio de bits alterados por mensaje crece de forma lineal para todos los algoritmos. En donde CRC tiende a mostrar un promedio ligeramente mayor, posiblemente esto es debido por la longitud de los mensajes o la forma en que se simula el ruido. La similitud entre las pruebas de 10K y 100K mensajes confirma la robustez y consistencia de la simulación.

Por otro lado, en las gráficas de "Tasa de éxito por algoritmo", el algoritmo de Hamming destaca con la mayor tasa de éxito, superando el 90%, esto es debido a su capacidad de corregir errores de un solo bit. Fletcher y CRC presentan tasas de éxito menores, ya que solo detectan errores y descartan cualquier mensaje alterado. Estos resultados demuestran la ventaja de los algoritmos de corrección en ambientes con errores simples, mientras que los de solo detección son más estrictos pero menos tolerantes a errores. Ahora bien, las gráficas de "Éxito vs Probabilidad de Ruido" muestran cómo la tasa de éxito disminuye al aumentar el ruido. Hamming mantiene la mejor resistencia, mientras que Fletcher y CRC caen rápidamente, ya que no pueden recuperar mensajes alterados. Esto resalta la importancia de la corrección de errores en canales ruidosos y la necesidad de elegir el algoritmo adecuado según el contexto de uso.

Adicionalmente, las gráficas de falsos positivos evidencian un comportamiento importante en el algoritmo de Hamming: a medida que aumenta la longitud de los mensajes enviados, crece también la probabilidad de que ocurran falsos positivos. Estos errores, aunque parciales en la mayoría de los casos, muestran que la corrección de un solo bit puede introducir vulnerabilidades cuando la cadena original es larga y el número de bits de paridad aumenta. Este hallazgo destaca la importancia de combinar la corrección con verificaciones adicionales para evitar aceptar como válidos mensajes corruptos.

Finalmente, los resultados permiten reflexionar sobre el impacto práctico de cada algoritmo. Hamming sería más apropiado en contextos donde los errores suelen ser aislados, como en canales con poco ruido. CRC-32, aunque no corrige, es muy usado en redes y almacenamiento debido a su simplicidad y confiabilidad en la detección. Fletcher, por su parte, ofrece un balance entre eficiencia y capacidad de detección, lo que lo hace útil en sistemas embebidos o hardware con recursos limitados. No obstante, el laboratorio también deja en claro que ningún algoritmo es completamente consistente: CRC-32 y Fletcher son vulnerables a colisiones, y Hamming a falsos positivos. La elección del algoritmo depende, entonces, de si se prioriza la detección eficiente o la corrección de errores, y de las condiciones reales del canal de comunicación.

Comentarios

A partir de la implementación de un cliente-servidor, se lograron observar diferencias claras entre detección y corrección en un entorno más cercano a la práctica real, donde se manejan grandes volúmenes de datos. Además, consideramos que la simulación de ruido fue útil para validar la robustez de cada algoritmo, aunque el modelo probabilístico utilizado es simplificado comparado con un canal real.

Conclusiones

- No existe un algoritmo universalmente mejor, ya que la elección depende de si se prioriza la corrección de errores (Hamming) o la detección eficiente (Fletcher y CRC-32)
- El código de Hamming sobresale en ambientes con errores simples, gracias a su capacidad de corrección de un bit, alcanzando tasas de éxito superiores al 90%.
- CRC-32 y Fletcher Checksum destacan en la detección, pero su falta de corrección los hace menos tolerantes a canales con ruido elevado.
- Los falsos positivos en Hamming aumentan con la longitud de los mensajes, lo que demuestra que su confiabilidad puede reducirse en mensajes muy largos.
- CRC-32, aunque muy usado en protocolos reales, puede presentar colisiones; por lo tanto, no es recomendable confiar únicamente en él para seguridad crítica.
- Fletcher ofrece un buen balance entre simplicidad y detección, pero al igual que CRC, requiere ser complementado con otros mecanismos si se busca alta confiabilidad.

Referencias

- CRC-32. (s. f.). En *Rosetta Code*. <https://rosettacode.org/wiki/CRC-32>
- *Fletcher's Checksum*. (s. f.).
<https://heasarc.gsfc.nasa.gov/docs/heasarc/ofwg/docs/general/checksum/node20.html>
- Mishra, N. (2017, 6 marzo). *Hamming Code in C and C++*. The Crazy Programmer.
<https://www.thecrazyprogrammer.com/2017/03/hamming-code-c.html>

Anexos

1. Enlace al repositorio: <https://github.com/paulabaal12/LAB2-REDES>