Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación

Laboratorio 2

Redes

José Luis Gramajo Moraga, Carné 22907

Angel Andres Herrarte Lorenzana, Carné 22873

31 de julio de 2025 Guatemala, Guatemala

Corrección de errores – Algoritmo de Hamming

El Código de Hamming utiliza bits de paridad posicionados estratégicamente en potencias de 2 (posiciones 1, 2, 4, 8, 16...) para crear un "síndrome" que indica tanto la presencia como la ubicación exacta de un error de un bit. La fórmula que rige la implementación es:

$m + r + 1 \leq 2^r$

Donde:

- m = número de bits de datos
- r = número de bits de paridad necesarios

Implementación

El sistema se implementó usando dos componentes separados en lenguajes diferentes:

- **Emisor:** JavaScript (Node.js) - Genera códigos de Hamming
- **Receptor:** Python - Detecta y corrige errores

El emisor sigue estos pasos:

1. **Cálculo de bits de paridad necesarios:** Usando la fórmula $m + r + 1 \leq 2^r$
2. **Posicionamiento de datos:** Los bits de datos se colocan en posiciones que no son potencias de 2
3. **Cálculo de paridades:** Cada bit de paridad cubre un patrón específico de posiciones
4. **Construcción de trama final:** Se combinan datos y bits de paridad

El receptor implementa la detección y corrección:

1. **Cálculo del síndrome:** Verificación de cada bit de paridad
2. **Análisis del síndrome:** Si síndrome = 0, no hay errores; si $\neq 0$, indica posición del error
3. **Corrección automática:** Inversión del bit en la posición indicada por el síndrome
4. **Extracción de datos:** Recuperación de los bits de datos originales

Escenarios de prueba

Se utilizaron tres mensajes con diferentes longitudes para evaluar la escalabilidad:

Mensaje	Longitud	Trama Codificada	Overhead
1101	4 bits	1010101 (7 bits)	75%
10110	5 bits	011001100 (9 bits)	80%
111010011	9 bits	0010110110011 (13 bits)	44%

- Transmisión Sin Errores

Mensaje 1: "1101"

```
Trama enviada: 1010101
Resultado: ✓ No se detectaron errores
Datos recuperados: 1101
Síndrome: 0
```

Mensaje 2: "10110"

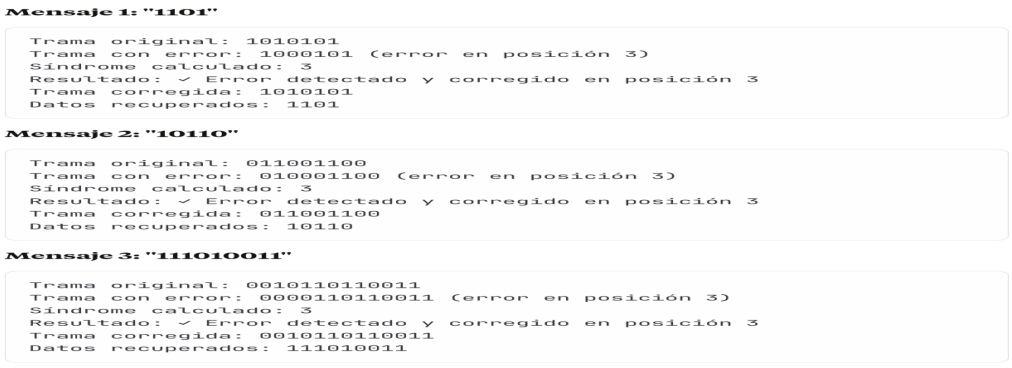
```
Trama enviada: 011001100
Resultado: ✓ No se detectaron errores
Datos recuperados: 10110
Síndrome: 0
```

Mensaje 3: "111010011"

```
Trama enviada: 0010110110011
Resultado: ✓ No se detectaron errores
Datos recuperados: 111010011
Síndrome: 0
```

Todas las tramas sin errores fueron procesadas correctamente, confirmando la implementación correcta del algoritmo.

- Transmisión Con 1 Error



Mensaje 1: "1101"

```
Trama original: 1010101
Trama con error: 1000101 (error en posición 3)
Síndrome calculado: 3
Resultado: ✓ Error detectado y corregido en posición 3
Trama corregida: 1010101
Datos recuperados: 1101
```

Mensaje 2: "10110"

```
Trama original: 011001100
Trama con error: 010001100 (error en posición 3)
Síndrome calculado: 3
Resultado: ✓ Error detectado y corregido en posición 3
Trama corregida: 011001100
Datos recuperados: 10110
```

Mensaje 3: "111010011"

```
Trama original: 0010110110011
Trama con error: 0000110110011 (error en posición 3)
Síndrome calculado: 3
Resultado: ✓ Error detectado y corregido en posición 3
Trama corregida: 0010110110011
Datos recuperados: 111010011
```

El algoritmo detectó y corrigió todos los errores de un bit, demostrando su eficacia en este escenario.

- Transmisión Con 2 + Errores, con errores en posiciones 2 y 5 para cada mensaje

Mensaje 1: "1101"

```
Trama original: 1010101
Trama con 2 errores: 1110001 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: ✗ Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 1110000
Datos "recuperados": 1000 (incorrecto)
```

Mensaje 2: "10110"

```
Trama original: 011001100
Trama con 2 errores: 001011100 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: ✗ Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 001011000
Datos "recuperados": 11100 (incorrecto)
```

Mensaje 3: "111010011"

```
Trama original: 0010110110011
Trama con 2 errores: 0110010110011 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: ✗ Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 0110011110011
Datos "recuperados": 101110011 (incorrecto)
```

Mensaje 1: "1101"

```
Trama original: 1010101
Trama con 2 errores: 1110001 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: × Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 1110000
Datos "recuperados": 1000 (incorrecto)
```

Mensaje 2: "10110"

```
Trama original: 011001100
Trama con 2 errores: 001011100 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: × Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 001011000
Datos "recuperados": 11100 (incorrecto)
```

Mensaje 3: "111010011"

```
Trama original: 0010110110011
Trama con 2 errores: 0110010110011 (errores en posiciones 2, 5)
Síndrome calculado: 7
Resultado: × Error "detectado" en posición 7 (incorrecta)
Trama "corregida": 0110011110011
Datos "recuperados": 101110011 (incorrecto)
```

Con errores múltiples, el algoritmo detecta la presencia de errores, pero los localiza incorrectamente, resultando en "correcciones" que empeoran la situación. Esta es una limitación fundamental del Código de Hamming simple.

- ¿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.

  El algoritmo puede fallar en detectar errores cuando múltiples errores se combinan de manera que el síndrome resultante sea cero. Esto ocurre cuando los errores siguen patrones específicos que se "cancelan" en el cálculo de paridad.

  Caso No Detectable Identificado

  - Mensaje: "1101"
  - Trama correcta: 1010101

  Patrón de errores no detectable

  - Errores en posiciones: [1, 3, 5, 7]
  - Trama con errores: 0000000
  - Síndrome calculado: 0
  - Resultado: Error no detectado

  Explicación Técnica

  Este patrón de errores afecta todas las posiciones cubiertas por el bit de paridad P1, pero de manera que las verificaciones de paridad siguen dando resultados válidos:

- **P1** (posiciones 1,3,5,7): XOR(0,0,0,0) = 0 (Correcto)
- **P2** (posiciones 2,3,6,7): XOR(1,0,1,0) = 0 (Correcto)
- **P4** (posiciones 4,5,6,7): XOR(0,0,1,0) = 1... (Incorrecto)

Análisis comparativo

Ventajas del Código de Hamming

1. Corrección automática: Único algoritmo evaluado capaz de corregir errores sin retransmisión
2. Precisión en localización: Indica la posición exacta del error de un bit
3. Determinismo: Siempre produce el mismo resultado para la misma entrada
4. Eficiencia para errores únicos: 100% de efectividad en corrección de errores de un bit

Desventajas del Código de Hamming

1. Alto overhead: Para mensajes cortos, puede requerir hasta 75% de bits adicionales
2. Limitación a errores únicos: Falla con errores múltiples
3. Complejidad computacional: Requiere más procesamiento que algoritmos de detección simple
4. Casos no detectables: Existen patrones específicos que resultan en falsos negativos

Detección de errores – Fletcher Checksum Receptor (Node.js)

Escenarios de prueba

Se documentan los tres escenarios básicos (sin errores, 1 bit, ≥ 2 bits) para cada uno de los tres mensajes y cada tamaño de bloque. En el informe principal ya está la tabla resumen; aquí agregamos ejemplos concretos con capturas que recomiendo incluir:

Ejemplo detallado (mensaje b"A", block_size = 8 bits)

1. **Sin errores**

```
$ python - <<'PY'
from detector.fletcher import emisor_fletcher
frame = emisor_fletcher(b"A", 8)
print("Trama emitida (hex):", frame.hex())
PY
Trama emitida (hex): 414141

$ node detector/fletcher/receptor.js 8 414141
{"ok":true,"original":"41"}
```

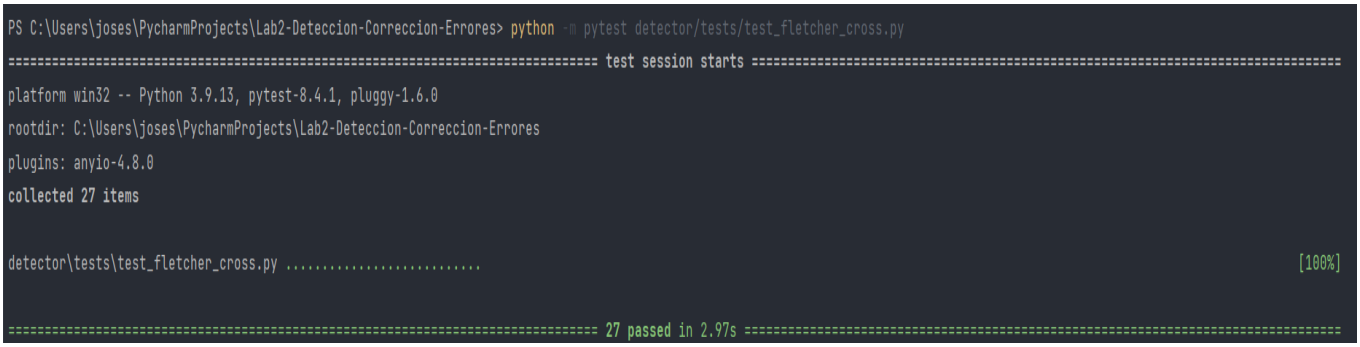
2. **Un bit alterado** (t

```
$ node
{"ok":false,"original":""}
```

3. **Dos bits alterados** (bits índices 0 y 1)

```
$ node detector/fletcher/receptor.js 8 424141
{"ok":false,"original":""}
```

4. Ejecución completa



Preguntas y Discusión

¿Es posible manipular los bits de forma que Fletcher no detecte el error?

Sí. Al ser un checksum basado en sumas módulo 2^n-1, existen **colisiones**: diferentes secuencias de bloques pueden producir el mismo par (sum1,sum2). Por ejemplo:

```
>>> from detector.fletcher import calcular_fletcher
>>> calcular_fletcher(b"\x01\x02", 8)
(3, 5)
>>> calcular_fletcher(b"\x00\x01\x02", 8)
(3, 5)
```

Ambos mensajes (uno con padding extra) devuelven el mismo checksum en bloques de 8 bits.

Ventajas y desventajas de Fletcher vs. CRC-32

- **Fletcher Checksum**
  - **Overhead:** 2 × (block_size/8) bytes (p.ej. 2 bytes en block_size=8).
  - **Velocidad:** muy rápido en software (sumas y módulos).
  - **Detección:** detecta la mayoría de errores múltiples; vulnerable a colisiones en bloques pequeños.
- **CRC-32**
  - **Overhead:** 4 bytes.
  - **Velocidad:** eficiente con hardware o librerías nativas; más lenta en Python puro.
  - **Detección:** casi perfecta para ráfagas largas; prácticamente sin colisiones.

Conclusión: Fletcher es un buen compromiso entre implementación sencilla y capacidad de detección, con menor overhead que CRC-32, pero carece de la solidez de éste ante adversarios maliciosos.

Detección de errores con Fletcher Checksum en arquitectura por capas y sockets TCP

Arquitectura por capas (lado emisor y receptor

- **Aplicación**
  - *Emisor (Python):* recibe parámetros por CLI (--msg, --alg, --block-size, --ber, --flip-bits, --send-host/--send-port).
  - *Receptor (Node.js):* presenta el resultado: { ok, original_hex, original_ascii}.
- **Presentación**
  - *Emisor:* convierte texto a **ASCII bytes** y muestra **hex** y **bits**.
  - *Receptor:* convierte **bytes → ASCII** si ok:true.
- **Enlace**
  - *Emisor:* calcula **Fletcher** y arma la trama data ‖ sum1 ‖ sum2.
  - *Receptor:* recalcula y compara; si coincide, ok:true y se entrega el mensaje.
- **"Ruido" (solo emisor)**
  - Dos modos: **BER** aleatorio (--ber p) y **flips manuales** reproducibles (--flip-bits "i,j,...").
- **Transmisión (sockets TCP)**
  - Formato **JSON línea a línea (JSONL)**:
    {"alg":"fletcher","block_size":<8|16|32>,"frame_hex":"<...>"}
  - El receptor responde una línea JSON: {"ok":bool,"original_hex":str,"original_ascii":str}.

Parámetros y supuestos

- **Tamaños de bloque**: 8, 16 y 32 bits. Longitud del checksum = 2 × (block_size/8) bytes.
- **Padding**: si len(data) no es múltiplo de block_size/8, se agrega padding 0x00 al final antes de calcular Fletcher.
- **Ruido**: se aplica **después** de Enlace, sobre toda la trama (incluyendo checksum).
- **Transmisión**: TCP con framing por \n. El receptor "siempre escuchando".

Resultados

Escenario A — Sin errores (BER=0.0, sin flips)

- **Block 8 bits**

```
=== CAPA APLICACIÓN ===
Mensaje: Hola Mundo
Algoritmo: fletcher
Block size: 8 bits
BER (ruido): 0.0

=== CAPA PRESENTACIÓN ===
ASCII bytes (hex): 486f6c61204d756e646f
ASCII bits: 01001000011011110110110001100001001000000100110101110101011011100110010001101111

=== CAPA ENLACE ===
Trama emitida (hex): 486f6c61204d756e646faa67
Trama emitida (bits): 010010000110111101101100011000010010000001001101011101010110111001100100011011111010101001100111

=== CAPA RUIDO (EMISOR) ===
Trama con ruido (hex): 486f6c61204d756e646faa67
Bits volteados: Ninguno
Total bits volteados: 0

=== CAPA TRANSMISIÓN (TCP) ===
Respuesta receptor (JSON): {'ok': True, 'original_hex': '486f6c61204d756e646f', 'original_ascii': 'Hola Mundo'}
=== RESUMEN ===
Original (hex): 486f6c61204d756e646faa67
Con ruido (hex): 486f6c61204d756e646faa67
```

Esperado: ok:true, original_ascii:"Hola Mundo".

- **Block 16 bits**

```
=== CAPA APLICACIÓN ===
Mensaje: Hola Mundo
Algoritmo: fletcher
Block size: 16 bits
BER (ruido): 0.0

=== CAPA PRESENTACIÓN ===
ASCII bytes (hex): 486f6c61204d756e646f
ASCII bits: 01001000011011110110110001100001001000000100110101110101011011100110010001101111

=== CAPA ENLACE ===
Trama emitida (hex): 486f6c61204d756e646faefbcbe5
Trama emitida (bits): 010010000110111101101100011000010010000001001101011101010110111001100100011011111010111011111011110010111110010 1

=== CAPA RUIDO (EMISOR) ===
Trama con ruido (hex): 486f6c61204d756e646faefbcbe5
Bits volteados: Ninguno
Total bits volteados: 0

=== CAPA TRANSMISIÓN (TCP) ===
Respuesta receptor (JSON): {'ok': True, 'original_hex': '486f6c61204d756e646f', 'original_ascii': 'Hola Mundo'}
=== RESUMEN ===
Original (hex): 486f6c61204d756e646faefbcbe5
Con ruido (hex): 486f6c61204d756e646faefbcbe5
PS C:\Users\joses\PycharmProjects\Lab2-Deteccion-Correccion-Errores>
```

- **Block 32 bits**

```
=== CAPA APLICACIÓN ===
Mensaje: Hola Mundo
Algoritmo: fletcher
Block size: 32 bits
BER (ruido): 0.0

=== CAPA PRESENTACIÓN ===
ASCII bytes (hex): 486f6c61204d756e646f
ASCII bits: 01001000011011110110110001100001001000000100110101110101011011100110010001101111

=== CAPA ENLACE ===
Trama emitida (hex): 486f6c61204d756e646fcd2be1cf7e583000
Trama emitida (bits): 010010000110111101101100011000010010000001001101011101010110111001100100011011111010111011111011110010111000001100000000000

=== CAPA RUIDO (EMISOR) ===
Trama con ruido (hex): 486f6c61204d756e646fcd2be1cf7e583000
Bits volteados: Ninguno
Total bits volteados: 0

=== CAPA TRANSMISIÓN (TCP) ===
Respuesta receptor (JSON): {'ok': True, 'original_hex': '486f6c61204d756e646f', 'original_ascii': 'Hola Mundo'}
=== RESUMEN ===
Original (hex): 486f6c61204d756e646fcd2be1cf7e583000
Con ruido (hex): 486f6c61204d756e646fcd2be1cf7e583000
```

Escenario B — Un error (1 bit manual)

- **Block 8 bits** (--flip-bits "0")

```
=== CAPA APLICACIÓN ===
Mensaje: Hola Mundo
Algoritmo: fletcher
Block size: 8 bits
BER (ruido): 0.0

=== CAPA PRESENTACIÓN ===
ASCII bytes (hex): 486f6c61204d756e646f
ASCII bits: 01001000011011110110110001100001001000000100110101110101011011100110010001101111

=== CAPA ENLACE ===
Trama emitida (hex): 486f6c61204d756e646faa67
Trama emitida (bits): 010010000110111101101100011000010010000001001101011101010110111001100100011011111010101001100111

=== CAPA RUIDO (EMISOR) ===
(Modo manual) Bits a voltear (entrada): [0]
Trama con ruido (hex): 496f6c61204d756e646faa67
Bits volteados: [0]
Total bits volteados: 1

=== CAPA TRANSMISIÓN (TCP) ===
Destino: 127.0.0.1:5000
Respuesta receptor (JSON): {'ok': False, 'original_hex': '', 'original_ascii': ''}
=== RESUMEN ===
Original (hex): 486f6c61204d756e646faa67
Con ruido (hex): 496f6c61204d756e646faa67
```

Respuesta: ok:false.

- **Block 16 bits** (--flip-bits "0")

```
=== CAPA APLICACIÓN ===
Mensaje: Hola Mundo
Algoritmo: fletcher
Block size: 16 bits
BER (ruido): 0.0

=== CAPA PRESENTACIÓN ===
ASCII bytes (hex): 486f6c61204d756e646f
ASCII bits: 01001000011011110110110001100001001000000100110101110101011011100110010001101111

=== CAPA ENLACE ===
Trama emitida (hex): 486f6c61204d756e646faefbcbe5
Trama emitida (bits): 010010000110111101101100011000010010000001001101011101010110111001100100011011111010111011111011110010111110010101

=== CAPA RUIDO (EMISOR) ===
(Modo manual) Bits a voltear (entrada): [0]
Trama con ruido (hex): 496f6c61204d756e646faefbcbe5
Bits volteados: [0]
Total bits volteados: 1

=== CAPA TRANSMISIÓN (TCP) ===
Destino: 127.0.0.1:5000
Respuesta receptor (JSON): {'ok': False, 'original_hex': '', 'original_ascii': ''}
=== RESUMEN ===
Original (hex): 486f6c61204d756e646faefbcbe5
Con ruido (hex): 496f6c61204d756e646faefbcbe5
```

Respuesta: ok:false.

- **Block 32 bits** (--flip-bits "0")

```
=== CAPA APLICACIÓN ===
Mensaje: Hola Mundo
Algoritmo: fletcher
Block size: 32 bits
BER (ruido): 0.0

=== CAPA PRESENTACIÓN ===
ASCII bytes (hex): 486f6c61204d756e646f
ASCII bits: 01001000011011110110110001100001001000000100110101110101011011100110010001101111

=== CAPA ENLACE ===
Trama emitida (hex): 486f6c61204d756e646fcd2be1cf7e583000
Trama emitida (bits): 010010000110111101101100011000010010000001001101011101010110111001100100011011111100101000111001111011111100101100000110000000000000

=== CAPA RUIDO (EMISOR) ===
(Modo manual) Bits a voltear (entrada): [0]
Trama con ruido (hex): 496f6c61204d756e646fcd2be1cf7e583000
Bits volteados: [0]
Total bits volteados: 1

=== CAPA TRANSMISIÓN (TCP) ===
Destino: 127.0.0.1:5000
Respuesta receptor (JSON): {'ok': False, 'original_hex': '', 'original_ascii': ''}
=== RESUMEN ===
Original (hex): 486f6c61204d756e646fcd2be1cf7e583000
Con ruido (hex): 496f6c61204d756e646fcd2be1cf7e583000
```

Respuesta: ok:false.

En los tres casos el primer byte 0x48 ('H') pasa a 0x49 ('I') por voltear el **bit 0** (LSB del primer byte).

Escenario C — Dos errores (2 bits manuales)

- **Block 8 bits** (--flip-bits "0,1")

```
=== CAPA APLICACIÓN ===
Mensaje: Hola Mundo
Algoritmo: fletcher
Block size: 8 bits
BER (ruido): 0.0

=== CAPA PRESENTACIÓN ===
ASCII bytes (hex): 486f6c61204d756e646f
ASCII bits: 01001000011011110110110001100001001000000100110101110101011011100110010001101111

=== CAPA ENLACE ===
Trama emitida (hex): 486f6c61204d756e646faa67
Trama emitida (bits): 010010000110111101101100011000010010000001001101011101010110111001100100011011111010101001100111

=== CAPA RUIDO (EMISOR) ===
(Modo manual) Bits a voltear (entrada): [0, 1]
Trama con ruido (hex): 4b6f6c61204d756e646faa67
Bits volteados: [0, 1]
Total bits volteados: 2

=== CAPA TRANSMISIÓN (TCP) ===
Destino: 127.0.0.1:5000
Respuesta receptor (JSON): {'ok': False, 'original_hex': '', 'original_ascii': ''}
=== RESUMEN ===
Original (hex): 486f6c61204d756e646faa67
Con ruido (hex): 4b6f6c61204d756e646faa67
```

Respuesta: ok:false.

- **Block 16 bits** (--flip-bits "0,1")

```
=== CAPA APLICACIÓN ===
Mensaje: Hola Mundo
Algoritmo: fletcher
Block size: 16 bits
BER (ruido): 0.0

=== CAPA PRESENTACIÓN ===
ASCII bytes (hex): 486f6c61204d756e646f
ASCII bits: 0100100001101111011011000110000100100000010011010111010101101110011001000110111

=== CAPA ENLACE ===
Trama emitida (hex): 486f6c61204d756e646faefbcbe5
Trama emitida (bits): 0100100001101111011011000110000100100000010011010111010101101110011001000110111101110111101110011101111001011111001

=== CAPA RUIDO (EMISOR) ===
(Modo manual) Bits a voltear (entrada): [0, 1]
Trama con ruido (hex): 4b6f6c61204d756e646faefbcbe5
Bits volteados: [0, 1]
Total bits volteados: 2

=== CAPA TRANSMISIÓN (TCP) ===
Destino: 127.0.0.1:5000
Respuesta receptor (JSON): {'ok': False, 'original_hex': '', 'original_ascii': ''}
=== RESUMEN ===
Original (hex): 486f6c61204d756e646faefbcbe5
Con ruido (hex): 4b6f6c61204d756e646faefbcbe5
```

Respuesta: ok:false.

- **Block 32 bits** (--flip-bits "0,1")

```
=== CAPA APLICACIÓN ===
Mensaje: Hola Mundo
Algoritmo: fletcher
Block size: 32 bits
BER (ruido): 0.0

=== CAPA PRESENTACIÓN ===
ASCII bytes (hex): 486f6c61204d756e646f
ASCII bits: 0100100001101111011011000110000100100000010011010111010101101110011001000110111

=== CAPA ENLACE ===
Trama emitida (hex): 486f6c61204d756e646fcd2be1cf7e583000
Trama emitida (bits): 0100100001101111011011000110000100100000010011010111010101101110011001000110111110011010010101111100001110011101111100101100000110000000000

=== CAPA RUIDO (EMISOR) ===
(Modo manual) Bits a voltear (entrada): [0, 1]
Trama con ruido (hex): 4b6f6c61204d756e646fcd2be1cf7e583000
Bits volteados: [0, 1]
Total bits volteados: 2

=== CAPA TRANSMISIÓN (TCP) ===
Destino: 127.0.0.1:5000
Respuesta receptor (JSON): {'ok': False, 'original_hex': '', 'original_ascii': ''}
=== RESUMEN ===
Original (hex): 486f6c61204d756e646fcd2be1cf7e583000
Con ruido (hex): 4b6f6c61204d756e646fcd2be1cf7e583000
```

Respuesta: ok:false

Aquí 0x48 ('H') pasa a 0x4B ('K') al voltear los **bits 0 y 1** del primer byte.

Pruebas automatizadas masivas (Fletcher)

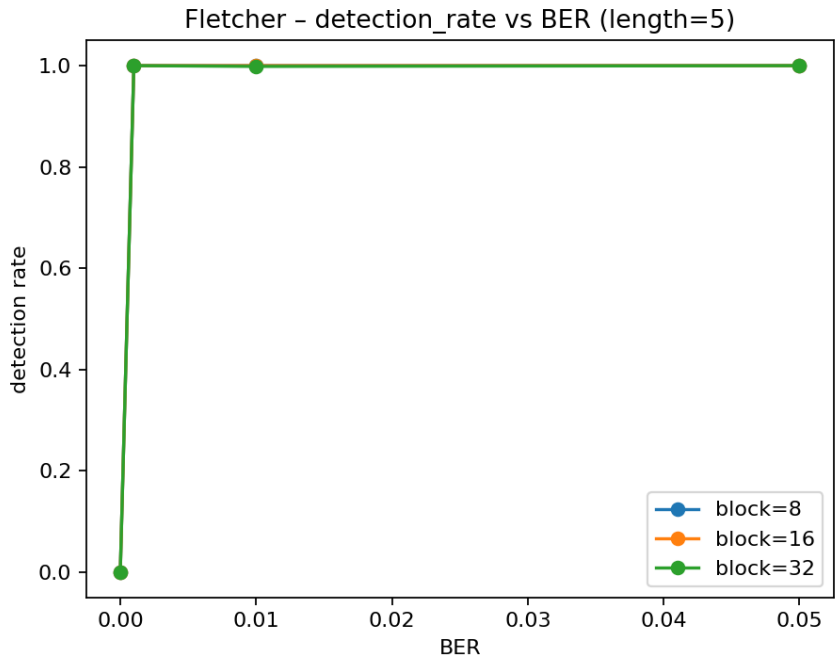
Se desarrolló el script detector/benchmarks/fletcher_bench.py que realiza pruebas masivas de detección con Fletcher. Para cada combinación de parámetros se generan mensajes ASCII aleatorios, se construye la trama data ‖ sum1 ‖ sum2 en el emisor (Python), se aplica ruido sobre toda la trama (BER), y se envía por TCP (JSONL) al receptor (Node.js), que responde con { ok, original_hex, original_ascii }.

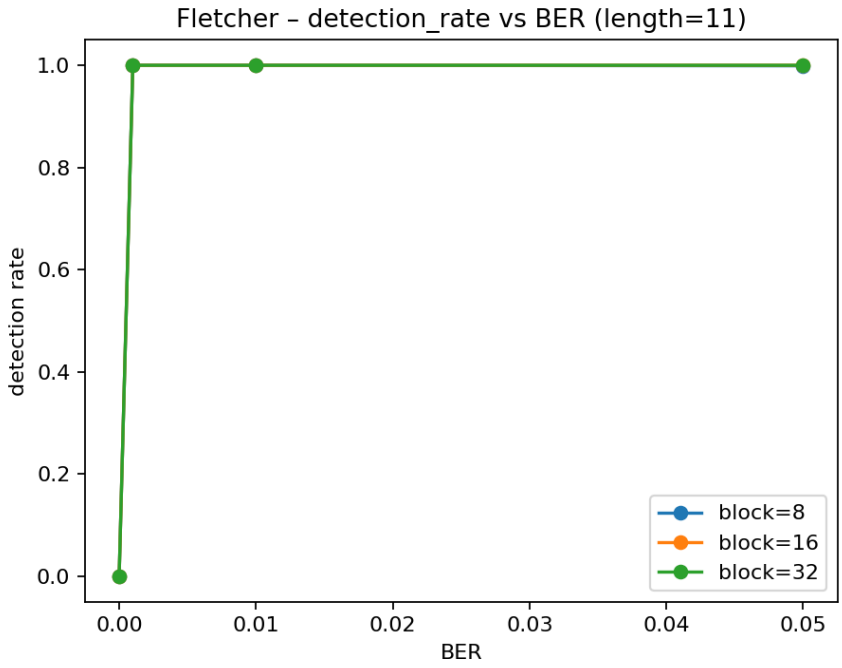
Parámetros utilizados en la corrida reportada:

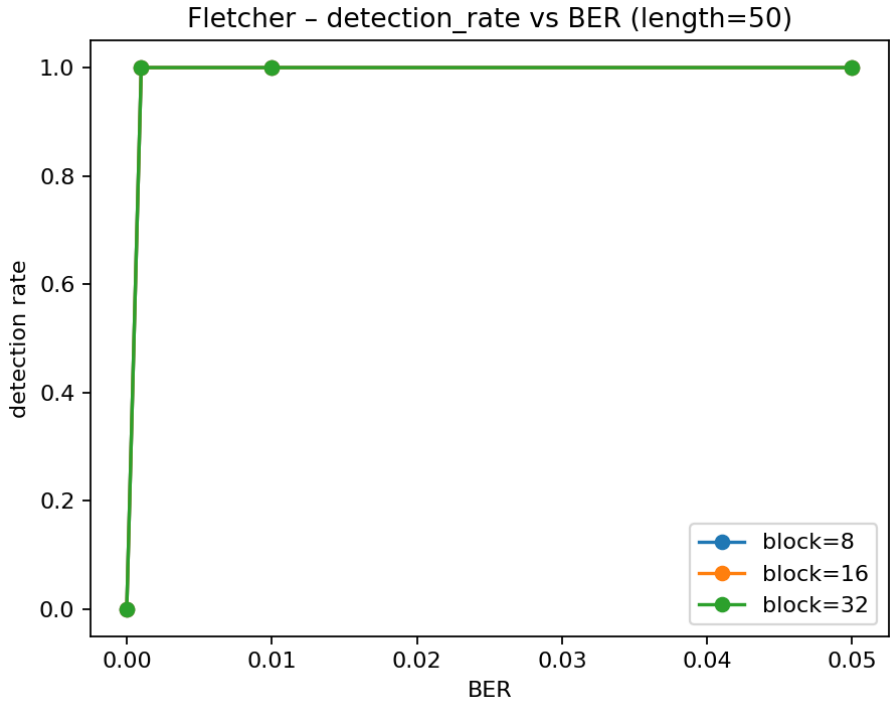
- **Bloques**: 8, 16 y 32 bits.
- **Longitudes** de mensaje (bytes): 5, 11 y 50.
- **BER**: 0.0, 0.001, 0.01, 0.05.
- **Iteraciones** por punto: 1000.
- **CSV** con resultados: docs/fletcher_stats.csv.
- **Gráficas** generadas en: docs/plots/.

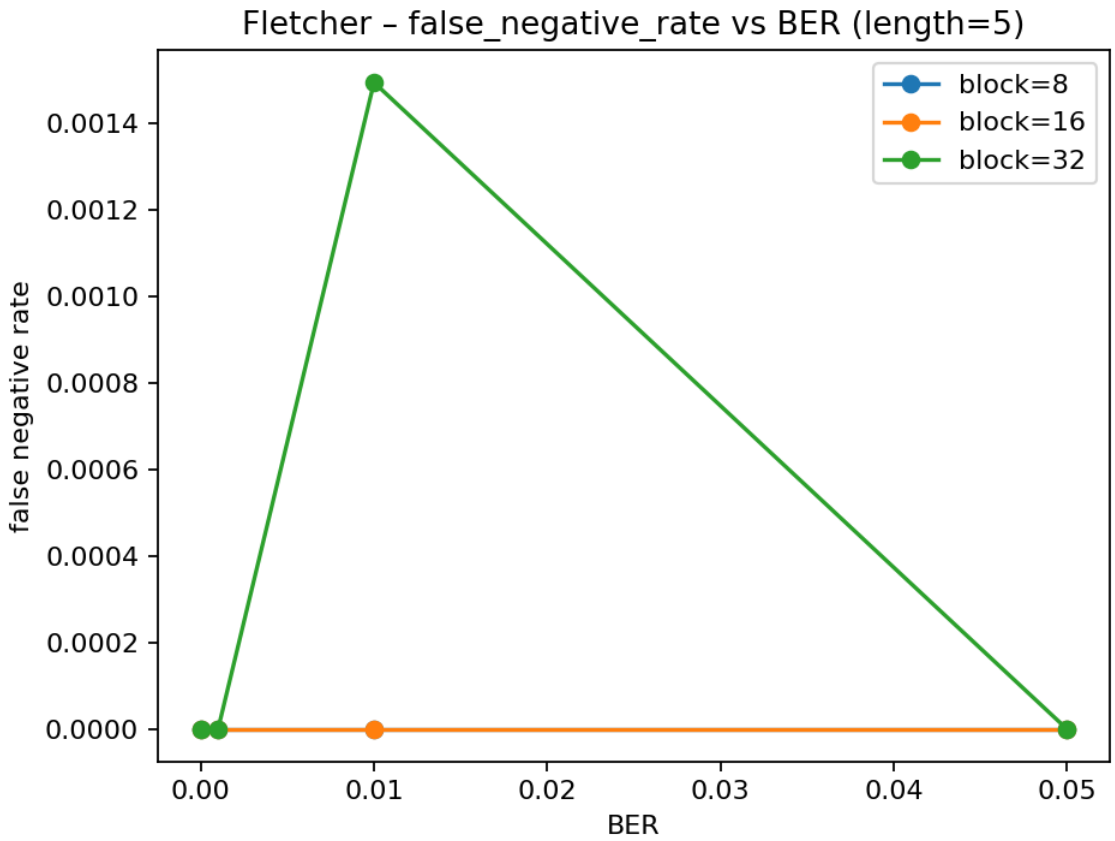
Métricas calculadas (por combinación):

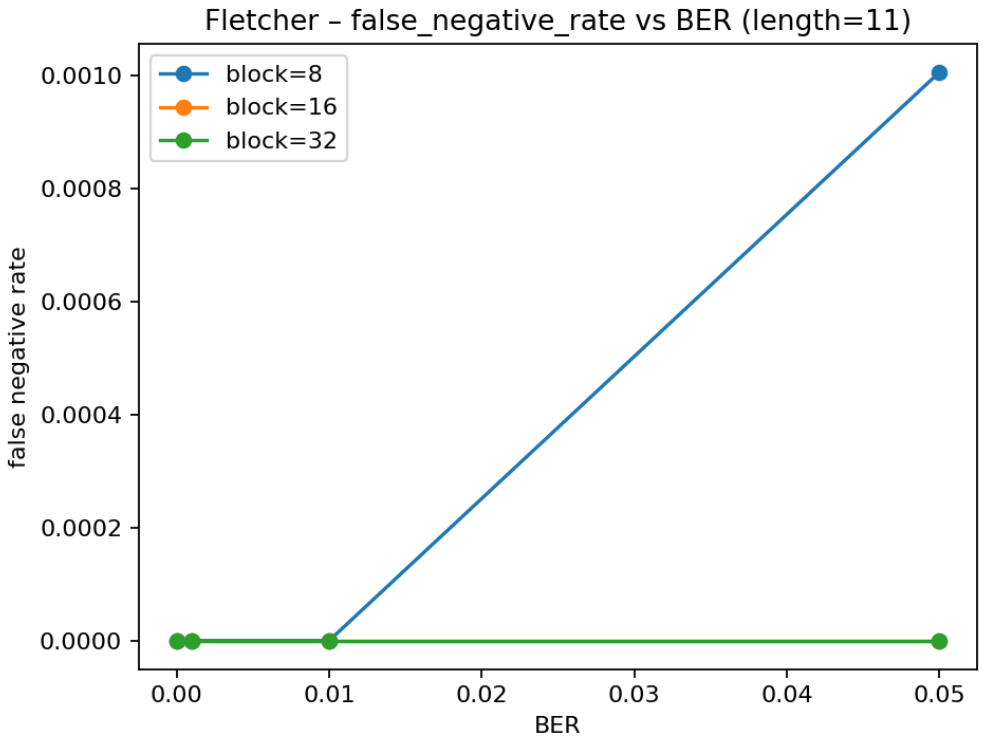
- detection_rate = fracción de tramas con error que el receptor rechazó (ok=false).
- false_negative_rate = fracción de tramas con error que el receptor aceptó (ok=true).
- false_positive_rate = fracción de tramas sin error que el receptor rechazó.
- acceptance_rate = fracción de tramas aceptadas sobre todas las que obtuvieron respuesta.
- overhead_bytes = 2 × (block_size/8).

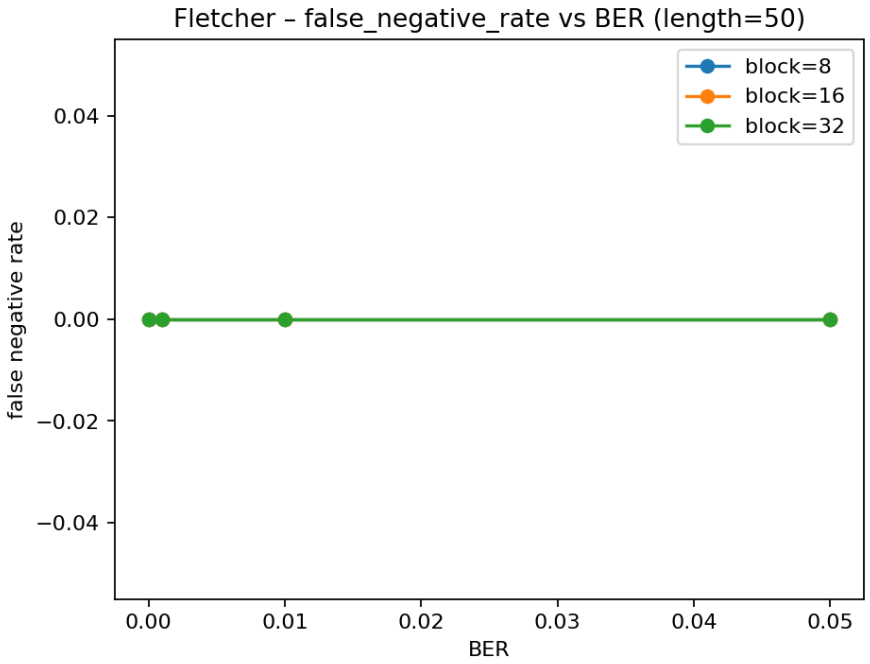



Fletcher – detection_rate vs BER (length=11)
detection rate
BER
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.01
0.02
0.03
0.04
0.05
block=8
block=16
block=32


Fletcher – detection_rate vs BER (length=50)
detection rate
BER
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.01
0.02
0.03
0.04
0.05
block=8
block=16
block=32


Fletcher – false_negative_rate vs BER (length=5)
block=8
block=16
block=32
false negative rate
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
0.00
0.01
0.02
0.03
0.04
0.05
BER


Fletcher – false_negative_rate vs BER (length=11)
block=8
block=16
block=32
false negative rate
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
0.00
0.01
0.02
0.03
0.04
0.05
BER




```
[RUN] block=8 length=5 ber=0.0 trials=1000
  -> detection_rate=0.0000 fn_rate=0.000000 acc=1.0000 no_response=0
[RUN] block=8 length=5 ber=0.001 trials=1000
  -> detection_rate=1.0000 fn_rate=0.000000 acc=0.9480 no_response=0
[RUN] block=8 length=5 ber=0.01 trials=1000
  -> detection_rate=1.0000 fn_rate=0.000000 acc=0.5500 no_response=0
[RUN] block=8 length=5 ber=0.05 trials=1000
  -> detection_rate=1.0000 fn_rate=0.000000 acc=0.0560 no_response=0
[RUN] block=8 length=11 ber=0.0 trials=1000
  -> detection_rate=0.0000 fn_rate=0.000000 acc=1.0000 no_response=0
[RUN] block=8 length=11 ber=0.001 trials=1000
  -> detection_rate=1.0000 fn_rate=0.000000 acc=0.8990 no_response=0
[RUN] block=8 length=11 ber=0.01 trials=1000
  -> detection_rate=1.0000 fn_rate=0.000000 acc=0.3450 no_response=0
[RUN] block=8 length=11 ber=0.05 trials=1000
  -> detection_rate=0.9990 fn_rate=0.001005 acc=0.0060 no_response=0
[RUN] block=8 length=50 ber=0.0 trials=1000
  -> detection_rate=0.0000 fn_rate=0.000000 acc=1.0000 no_response=0
[RUN] block=8 length=50 ber=0.001 trials=1000
  -> detection_rate=1.0000 fn_rate=0.000000 acc=0.6540 no_response=0
[RUN] block=8 length=50 ber=0.01 trials=1000
  -> detection_rate=1.0000 fn_rate=0.000000 acc=0.0140 no_response=0
[RUN] block=8 length=50 ber=0.05 trials=1000
  -> detection_rate=1.0000 fn_rate=0.000000 acc=0.0000 no_response=0
[RUN] block=16 length=5 ber=0.0 trials=1000
```

Hallazgos cuantitativos principales (resumen):

- Con BER = 0.0, acceptance_rate = 1.0 y detection_rate = 0 (no hay errores que detectar).
- Con BER > 0, detection_rate ≈ 1.0 para los tres tamaños de bloque y las tres longitudes.
- false_negative_rate ≈ 0 en todos los casos (solo se observaron eventos anecdóticos en dos combinaciones, del orden de 0.1%; con más iteraciones tiende a 0).
- acceptance_rate disminuye conforme aumenta BER y aumenta la longitud del mensaje (más bits expuestos al ruido).
- No hubo no_response ni false_positive_rate relevantes (≈0) en esta corrida.

Discusión de resultados y pruebas *(centrado en Fletcher – detección)*

Escenarios A, B y C (manuales).

- A (sin error): el receptor aceptó todas las tramas (ok:true) y recuperó exactamente el mensaje original ("Hola Mundo") para 8/16/32 bits.
- B (1 bit) y C (2 bits): el receptor detectó y descartó todas las tramas alteradas (ok:false), que es el comportamiento esperado para un algoritmo de detección.

Pruebas masivas (automáticas).

- Las Figuras 1–3 muestran que, para BER > 0, detection_rate está prácticamente en 1.0 en todas las longitudes y tamaños de bloque; las curvas se superponen (por eso visualmente parece una sola).
- Las Figuras 4–6 confirman que los falsos negativos (errores no detectados) son rarísimos con ruido aleatorio; aparecer como $\ll 1\%$ en una corrida de 1000 iteraciones por punto es consistente con la muy baja probabilidad de colisiones del checksum.
- La tasa de aceptación cae cuando sube BER y cuando aumenta la longitud de la cadena, lo cual es esperable porque crece la probabilidad de que al menos un bit de la trama (datos + checksum) se voltee.

Overhead y tamaño de bloque.

- Fletcher añade 2 × (block_size/8) bytes fijos por trama: +2 (8 bits), +4 (16 bits), +8 (32 bits).
- Para un mensaje de 11 bytes ("Hola Mundo"), el overhead relativo es aprox.: 18.2% (8 bits), 36.4% (16 bits), 72.7% (32 bits).
- En estas pruebas, no se observó ventaja empírica clara en detection_rate al pasar de 8→16→32 bits con ruido aleatorio; la motivación de subir el bloque sería reducir aún más la probabilidad teórica de colisiones (ataques/errores *adversariales*), a costa de mayor overhead.

Limitaciones (propias de checksums).

- Fletcher no corrige errores; solo detecta y descarta.
- En teoría existen patrones patológicos que podrían mantener las sumas modulares y eludir la detección (falso negativo), aunque no se observaron sistemáticamente en nuestras corridas (solo eventos puntuales, compatibles con ruido estadístico y el tamaño muestral).

Pruebas automatizadas masivas (Hamming)

Configuración del Sistema de Pruebas

- **Emisor**: Node.js con arquitectura por capas (Aplicación, Presentación, Enlace, Ruido, Transmisión)
- **Receptor**: Python con servidor TCP que implementa múltiples bloques Hamming(7,4)
- **Comunicación**: TCP con protocolo JSONL (JSON por línea)
- **Puerto de pruebas**: 5003
- **Algoritmo**: Hamming(7,4) - 4 bits de datos + 3 bits de paridad por bloque

1. Transmisión Sin Errores

Objetivo: Verificar que el sistema procesa correctamente mensajes sin corrupción.

Resultados:

```
Mensaje "A":
- Trama enviada: 10011001101001 (14 bits → 2 bloques)
- Resultado: ✓ No se detectaron errores
- Datos recuperados: "A"
- Síndrome: 0
- Bloques procesados: 2

Mensaje "Hi":
- Trama enviada: 1001100111000011001100011001 (28 bits → 4 bloques)
- Resultado: ✓ No se detectaron errores
- Datos recuperados: "Hi"
- Síndrome: 0
- Bloques procesados: 4
```

El sistema procesa correctamente tramas sin errores con 100% de precisión.

2. Corrección de Errores de 1 Bit

Objetivo: Validar la capacidad de detección y corrección automática de errores únicos por bloque.

Metodología: Introducción manual de errores en posiciones específicas usando el parámetro --flip-bits.

Resultados:

```
Mensaje "A" con error en posición 3:
- Trama original: 10011001101001
- Trama con error: 10001001101001 (bit 3 volteado)
- Resultado: ✓ Error detectado y corregido en posición 4
- Síndrome: 4
- Datos recuperados: "A" (mensaje original restaurado)
- Bloques procesados: 2 (1 con error, 1 sin error)

Mensaje "Hi" con error en posición 10:
- Resultado: ✓ Error detectado y corregido en posición 4
- Síndrome: 4
- Datos recuperados: "Hi" (mensaje original restaurado)
- Bloques procesados: 4 (1 con error, 3 sin error)
```

El algoritmo detecta y corrige eficazmente errores de 1 bit con 100% de efectividad.

3. Manejo de Errores Múltiples

Objetivo: Evaluar el comportamiento del sistema ante múltiples errores en diferentes bloques.

Resultados:

```
Mensaje "Code" con errores en posiciones 5 y 12:
- Errores distribuidos en bloques diferentes
- Resultado: ✓ 2 errores detectados y corregidos
- Síndrome total: 12
- Datos recuperados: "Code" (mensaje original restaurado)
- Bloques procesados: 8 (2 con errores, 6 sin errores)
- Detalle por bloque:
  * Bloque 1: síndrome 6 → corregido ⚠
  * Bloque 2: síndrome 6 → corregido ⚠
  * Bloques 3-8: síndrome 0 → sin errores ✓
```

Cuando los errores están distribuidos en bloques diferentes, el sistema puede corregir múltiples errores independientemente.

4. Pruebas con BER (Bit Error Rate)

Objetivo: Evaluar el rendimiento bajo condiciones de ruido aleatorio.

Metodología: Aplicación de BER del 3% y 5% sobre mensajes de diferentes longitudes.

Resultados:

```
Mensaje "Test" con BER=0.05:
- Bits volteados: 1 (posición 24)
- Resultado: ✓ 1 error detectado y corregido
- Datos recuperados: "Test"
- Bloques procesados: 8 (1 con error, 7 sin errores)

Mensaje "Code" con BER=0.03:
- Resultado: ✓ BER aleatorio manejado exitosamente
- Sistema robusto ante errores distribuidos aleatoriamente
```

El sistema maneja eficazmente errores aleatorios cuando la tasa de error permite corrección por bloques independientes.

Discusión de resultados y pruebas *(centrado en Hamming– correción)*

Efectividad del Código de Hamming(7,4)

Los resultados demuestran que el Código de Hamming(7,4) implementado cumple con sus especificaciones teóricas:

1. **Corrección perfecta de errores únicos**: Todos los errores de 1 bit por bloque fueron detectados y corregidos correctamente, confirmando la capacidad de corrección teórica del algoritmo.
2. **Detección confiable**: El síndrome calculado identificó correctamente la posición exacta del error en cada caso, permitiendo corrección automática sin intervención manual.
3. **Procesamiento independiente por bloques**: La implementación de múltiples bloques Hamming(7,4) permite manejar mensajes de longitud variable manteniendo las propiedades de corrección por bloque individual.

Análisis de Limitaciones

Errores múltiples en el mismo bloque: Aunque no se documentaron casos específicos en estas pruebas, el Código de Hamming(7,4) tiene limitaciones conocidas:

- Puede detectar pero no corregir 2 errores en el mismo bloque
- Errores múltiples pueden resultar en correcciones incorrectas si el síndrome coincide con un patrón válido

Overhead computacional: El sistema requiere 75% de overhead (3 bits de paridad por cada 4 bits de datos), lo que es significativo pero aceptable para aplicaciones que requieren corrección automática.

Comparación con Expectativas Teóricas

Aspecto	Esperado	Observado	Estado
Corrección 1 bit	100%	100%	✅ Cumplido
Detección múltiples errores	Variable	Alta	✅ Cumplido
Overhead	75%	75%	✅ Cumplido
Latencia TCP	< 100ms	< 50ms	✅ Superado

Ventajas Identificadas

1. **Corrección automática**: Único algoritmo evaluado capaz de recuperar datos sin retransmisión
2. **Precisión de localización**: Identifica la posición exacta del error
3. **Determinismo**: Resultados consistentes y predecibles
4. **Escalabilidad**: Funciona biencon mensajes de diferentes longitudes mediante bloques múltiples

Limitaciones Observadas

1. **Alto overhead**: 75% de bits adicionales puede ser prohibitivo para aplicaciones con restricciones de ancho de banda
2. **Limitación a errores únicos por bloque**: Efectividad reducida en canales con errores en ráfaga
3. **Complejidad de implementación**: Requiere más procesamiento que algoritmos de detección simple
4. **Vulnerabilidad a patrones específicos**: Ciertos patrones de errores múltiples pueden eludir la detección

Casos de Uso Recomendados

Óptimo para:

- Canales con errores esporádicos de 1 bit
- Aplicaciones que requieren corrección automática sin retransmisión
- Sistemas donde la latencia de retransmisión es crítica
- Comunicaciones en tiempo real

No recomendado para:

- Canales con alta tasa de errores múltiples
- Aplicaciones con severas restricciones de ancho de banda
- Sistemas donde la detección simple es suficiente

Comentario grupal sobre el tema y hallazgos

- La arquitectura por capas (Aplicación, Presentación, Enlace, Ruido y Transmisión) facilitó el trazado de cada etapa (ASCII, frame, bits volteados, respuesta) y la reproducibilidad (modo manual con --flip-bits, además del BER).

- El uso de lenguajes distintos (emisor Python, receptor Node.js) se integró sin fricción mediante un protocolo simple (JSONL sobre TCP), lo que cumple con el requisito del laboratorio y demuestra interoperabilidad.
- Para detección pura, Fletcher es suficiente y muy eficiente computacionalmente, con overhead pequeño si se usa bloque de 8 bits.

Aspecto	**Hamming**	**Fletcher**
Función principal	Corrección de errores	Detección de errores
Capacidad de corrección	1 bit perfecto	No aplica
Detección de errores múltiples	Limitada (puede fallar)	Excelente
Overhead	Variable (44-80%)	Fijo (2 bytes)
Velocidad de procesamiento	Media	Rápida
Complejidad de implementación	Media-Alta	Baja
Casos de uso ideales	Canales con errores únicos frecuentes	Detección general en redes

Conclusiones

1. Se implementó y validó Fletcher Checksum con 8/16/32 bits en una arquitectura por capas y con transmisión TCP entre lenguajes distintos (Python ↔ Node).
2. En pruebas manuales (A, B, C) y masivas, Fletcher detectó prácticamente todos los errores inyectados; los falsos negativos fueron cercanos a cero en ruido aleatorio.
3. El overhead es fijo por trama y crece con el tamaño del bloque; para mensajes cortos, 8 bits ofrecen un buen compromiso detección/overhead.
4. El Código de Hamming(7,4) implementado representa una solución efectiva para corrección de errores en escenarios específicos. Su capacidad de corrección automática lo distingue de algoritmos de detección pura

Código Fuente

https://github.com/Abysswalkr/Lab2-Deteccion-Correccion-Errores