

Laboratorio 2

Esquemas de detección y corrección de errores

<u>Laboratorio 2</u>	1
<u>Esquemas de detección y corrección de errores</u>	1
<u>I. Contexto</u>	1
<u>II. Resultados primera parte</u>	1
<u>a. Códigos de Hamming</u>	1
<u>b. Fletcher checksum</u>	3
<u>III. Implementación de la parte 2</u>	7
<u>a. Descripción general</u>	7
<u>b. Pruebas y Resultados</u>	8
<u>IV. GitHub</u>	10
<u>V. Discusión</u>	10
<u>VI. Conclusiones</u>	10
<u>VII. Referencias</u>	10

I. Contexto

Para realizar el laboratorio decidimos trabajar con los esquemas:

- Para corrección de errores **Códigos de Hamming**
- Para detección de errores **Fletcher checksum**

Además, decidimos implementar el emisor Java y el receptor con Python.

El código lo pueden encontrar en el repositorio:

https://github.com/Fabiola-cc/Correccion_Deteccion

II. Resultados primera parte

a. Códigos de Hamming

- Sin errores

Mensaje de prueba: 1010; 1011010

el canal, el síndrome imita un error simple; el decodificador corrige “un” bit y aterriza en otra palabra de código válida

```
Ingrese el mensaje recibido (con bits de paridad): 0100000
Mensaje recibido: 0100000
Longitud total del mensaje: 7
Bits de paridad detectados: 3

Matriz del mensaje recibido:
Posición: 1 2 3 4 5 6 7
Bit:      0 1 0 0 0 0 0

Verificando integridad del mensaje...
Posiciones de paridad: [1, 2, 4]

Verificando P1 (posición 1):
Posiciones que tienen '1' en el bit 1: [1, 3, 5, 7]
Bits a evaluar: 0 0 0 0
Resultado XOR (síndrome S1): 0

Verificando P2 (posición 2):
Posiciones que tienen '1' en el bit 2: [2, 3, 6, 7]
Bits a evaluar: 1 0 0 0
Resultado XOR (síndrome S2): 1

Verificando P3 (posición 4):
Posiciones que tienen '1' en el bit 3: [4, 5, 6, 7]
Bits a evaluar: 0 0 0 0
Resultado XOR (síndrome S3): 0

Síndrome completo: 010
Síndrome en decimal: 2

⚠️ SE DETECTÓ Y CORREGIÓ 1 ERROR
Error en la posición: 2
Mensaje corregido: 0000000
Mensaje original: 0000
```

- 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)

Durante las pruebas observé que, comparado con otros métodos, el código de Hamming ofrece la ventaja de poca redundancia (solo agrega algunos bits extra según la longitud del mensaje), además de que la implementación es sencilla y la corrección de un solo error es automática, lo cual lo hace útil en ambientes con ruido bajo. Una desventaja clara es que no detecta todos los errores múltiples: en mis pruebas se vio que al ocurrir dos errores simultáneos el receptor podía interpretarlo como un único error y corregirlo mal, entregando un mensaje falso sin advertencia. Esto muestra que en canales muy ruidosos no escala bien en comparación con CRC-32 o códigos convolucionales.

b. Fletcher checksum

- Sin errores

Usando Fletcher-8

Mensaje de prueba: 10110010

```
checksum_emisor.java
public class checksum_emisor {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // Inicializar recursos
        checksum_emisor emisor = new checksum_emisor(eleccion);
        System.out.println(String.format("Genial, estamos usando Fletcher-8. Ahora escribe el mensaje que quieres enviar (i.e.: 110101):"));
        String mensaje = sc.nextLine();
        // Calcular checksum
        int checksum = emisor.calcularChecksum(mensaje);
        System.out.println("El resultado de checksum es: " + checksum);
        System.out.println("El mensaje para el receptor: " + mensaje + checksum);
    }
}

checksum_receptor.py
class checksum_receptor:
    def __init__(self, tipo):
        self.tipo = tipo
        self.mensaje = ""
        self.checksum = ""
        self.mensaje_completo = ""

    def verificar_checksum(self, mensaje_recibido):
        # dividir mensaje y checksum
        mensaje_size = len(mensaje_recibido) - self.tipo
        mensaje = mensaje_recibido[:mensaje_size]
        checksum_recibido = mensaje_recibido[mensaje_size:]
        checksum_comparar = int(mensaje_recibido[-self.tipo:], 2)

        # verificar checksum
        if self.verificar_checksum(mensaje, checksum_recibido) == checksum_comparar:
            return "No se detectaron errores."
        else:
            return "El mensaje es: " + mensaje
```

```
PS C:\Users\Fabi\Documents\U\Github\Correccion_Deteccion> cd 'C:\Users\Fabi\Documents\U\Github\Correccion_Deteccion'; & 'C:\Program Files\Eclipse Adoptium\jdk-17.0.6.1-hotspot\bin\java.exe' -XX:+ShowCodeDetailsInExceptionMessages -cp 'C:\Users\Fabi\AppData\Roaming\Code\User\workspaceStorage\66c52864ee3688d08be9f5182bc4f787\redhat-javajdt_ws\Correccion_Deteccion_afc5e93\bin' 'checksum_emisor'
Bienvenido. Esta es una simulación de Checksum Fletcher

Elige el tipo con el que quieres trabajar. (Escribe 1, 2 o 3)
1. Fletcher-8
2. Fletcher-16
3. Fletcher-32
1

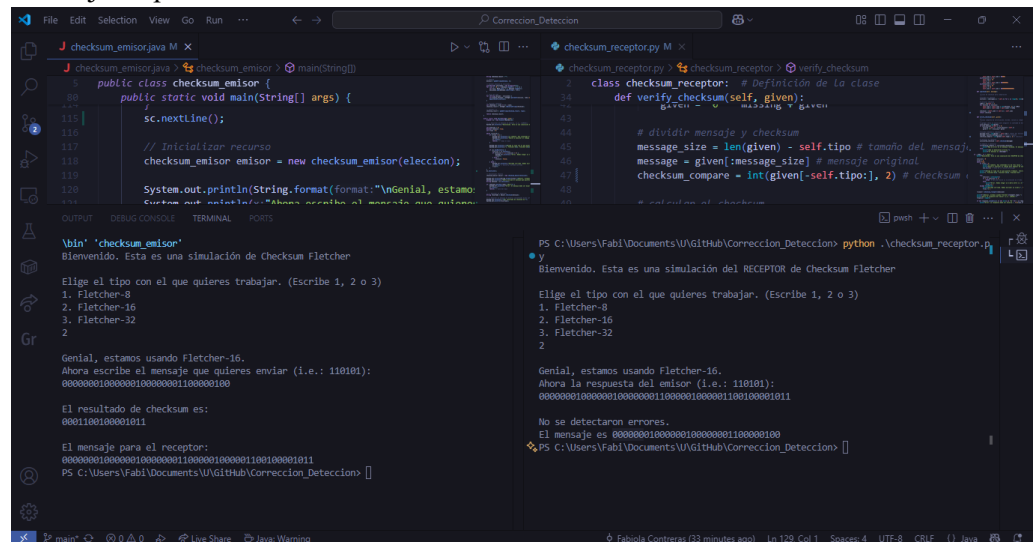
Genial, estamos usando Fletcher-8.
Ahora escribe el mensaje que quieres enviar (i.e.: 110101):
10110010

El resultado de checksum es:
11001110

El mensaje para el receptor:
1011001011001110
PS C:\Users\Fabi\Documents\U\Github\Correccion_Deteccion>
```

Usando Fletcher-16

Mensaje de prueba: 00000001000000100000001100000100

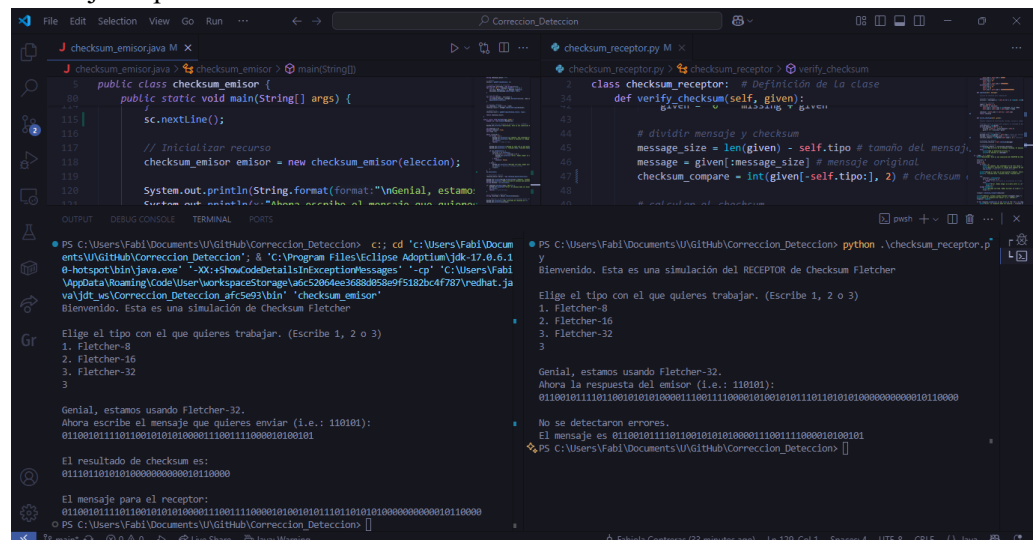


```
checksum_emisor.java
public class checksum_emisor {
    public static void main(String[] args) {
        // ...
        // Inicializan recurso
        checksum_emisor emisor = new checksum_emisor(eleccion);
        System.out.println(String.format(format:"\nGenial, estamos usando Fletcher-16. Ahora escribe el mensaje que quieres enviar (i.e.: 110101):
        00000001000000100000001100000100
        El resultado de checksum es:
        00011001000001011
        El mensaje para el receptor:
        0000000100000010000000110000010000011001000001011
        PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion>

checksum_receptor.py
class checksum_receptor:
    def verify_checksum(self, given):
        # dividir mensaje y checksum
        message_size = len(given) - self.tipo # tamaño del mensaje
        message = given[:message_size] # mensaje original
        checksum_compare = int(given[-self.tipo:], 2) # checksum
        # ...
        PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion> python .\checksum_receptor.py
        Bienvenido. Esta es una simulación del RECEPTOR de Checksum Fletcher
        Elige el tipo con el que quieres trabajar. (Escribe 1, 2 o 3)
        1. Fletcher-8
        2. Fletcher-16
        3. Fletcher-32
        2
        Genial, estamos usando Fletcher-16.
        Ahora la respuesta del emisor (i.e.: 110101):
        0000000100000010000000110000010000011001000001011
        No se detectaron errores.
        El mensaje es 0000000100000010000000110000010000011001000001011
        PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion>
```

Usando Fletcher-32

Mensaje de prueba: 011001011110110010101010000111001111000010100101



```
checksum_emisor.java
public class checksum_emisor {
    public static void main(String[] args) {
        // ...
        // Inicializan recurso
        checksum_emisor emisor = new checksum_emisor(eleccion);
        System.out.println(String.format(format:"\nGenial, estamos usando Fletcher-32. Ahora escribe el mensaje que quieres enviar (i.e.: 110101):
        011001011110110010101010000111001111000010100101
        El resultado de checksum es:
        0111010101010000000000010110000
        El mensaje para el receptor:
        01100101111011001010100001110011110000101001010000
        PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion>

checksum_receptor.py
class checksum_receptor:
    def verify_checksum(self, given):
        # dividir mensaje y checksum
        message_size = len(given) - self.tipo # tamaño del mensaje
        message = given[:message_size] # mensaje original
        checksum_compare = int(given[-self.tipo:], 2) # checksum
        # ...
        PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion> python .\checksum_receptor.py
        Bienvenido. Esta es una simulación del RECEPTOR de Checksum Fletcher
        Elige el tipo con el que quieres trabajar. (Escribe 1, 2 o 3)
        1. Fletcher-8
        2. Fletcher-16
        3. Fletcher-32
        3
        Genial, estamos usando Fletcher-32.
        Ahora la respuesta del emisor (i.e.: 110101):
        01100101111011001010100001110011110000101001010000
        No se detectaron errores.
        El mensaje es 0110010111101100101010000111001111000010100101
        PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion>
```

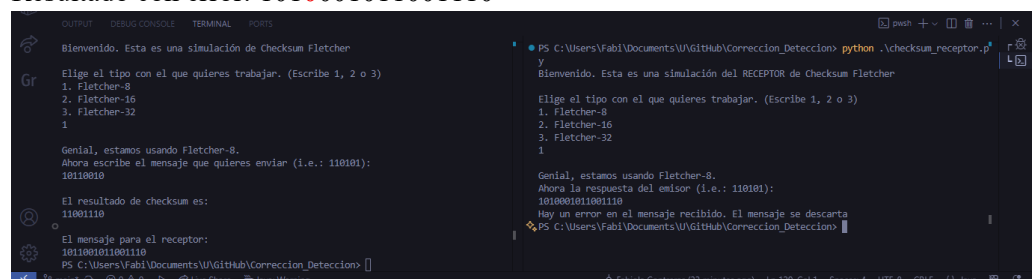
- Un error

Usando Fletcher-8

Mensaje de prueba: 10110010

Resultado: 1011001011001110

Resultado con error: 1010001011001110



```
checksum_emisor.java
public class checksum_emisor {
    public static void main(String[] args) {
        // ...
        // Inicializan recurso
        checksum_emisor emisor = new checksum_emisor(eleccion);
        System.out.println(String.format(format:"\nGenial, estamos usando Fletcher-8. Ahora escribe el mensaje que quieres enviar (i.e.: 110101):
        10110010
        El resultado de checksum es:
        1011001011001110
        El mensaje para el receptor:
        1011001011001110
        PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion>

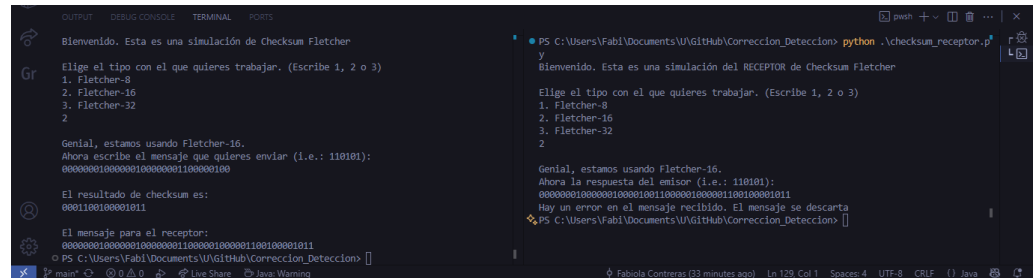
checksum_receptor.py
class checksum_receptor:
    def verify_checksum(self, given):
        # dividir mensaje y checksum
        message_size = len(given) - self.tipo # tamaño del mensaje
        message = given[:message_size] # mensaje original
        checksum_compare = int(given[-self.tipo:], 2) # checksum
        # ...
        PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion> python .\checksum_receptor.py
        Bienvenido. Esta es una simulación del RECEPTOR de Checksum Fletcher
        Elige el tipo con el que quieres trabajar. (Escribe 1, 2 o 3)
        1. Fletcher-8
        2. Fletcher-16
        3. Fletcher-32
        1
        Genial, estamos usando Fletcher-8.
        Ahora la respuesta del emisor (i.e.: 110101):
        1011001011001110
        Hay un error en el mensaje recibido. El mensaje se descarta
        PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion>
```

Usando Fletcher-16

Mensaje de prueba: 00000001000000100000001100000100

Resultado: 000000010000001000000011000001000001100100001011

con error: 000000010000001000010011000001000001100100001011



Usando Fletcher-32

Mensaje de prueba: 011001011110110010101010000111001111000010100101

Resultado:

0110010111101100101010100001110011110000101001010111011010101000

0000000010110000

con error:

0110010111101000101010100001110011110000101001010111011010101000

0000000010110000



- Dos errores

Usando Fletcher-8

Mensaje de prueba: 10110010

Resultado: 1011001011001110

Resultado con error: 1010001111001110



Usando Fletcher-16

Mensaje de prueba: 00000001000000100000001100000100

Resultado: 000000010000001000000011000001000001100100001011

con error: 000000010000001000010101000001000001100100001011



```
Gr
Bienvenido. Esta es una simulación de Checksum Fletcher
Elige el tipo con el que quieres trabajar. (Escribe 1, 2 o 3)
1. Fletcher-8
2. Fletcher-16
3. Fletcher-32
2

Genial, estamos usando Fletcher-16.
Ahora escribe el mensaje que quieres enviar (i.e.: 110101):
00000001000000100000001100000100

El resultado de checksum es:
0001100100001011

El mensaje para el receptor:
000000010000001000000011000001000001100100001011
PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion>
```

Usando Fletcher-32

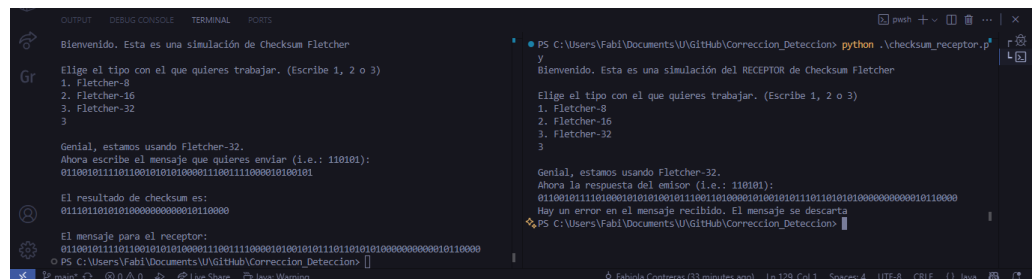
Mensaje de prueba: 011001011110110010101010000111001111000010100101

Resultado:

0110010111101100101010100001110011110000101001010111011010101000
0000000010110000

con error:

0110010111101000101010100101110011010000101001010111011010101000
0000000010110000



```
Gr
Bienvenido. Esta es una simulación de Checksum Fletcher
Elige el tipo con el que quieres trabajar. (Escribe 1, 2 o 3)
1. Fletcher-8
2. Fletcher-16
3. Fletcher-32
3

Genial, estamos usando Fletcher-32.
Ahora escribe el mensaje que quieres enviar (i.e.: 110101):
011001011110110010101010000111001111000010100101

El resultado de checksum es:
0111010101010000000000010110000

El mensaje para el receptor:
01100101111011001010100001110011110000101001010101010100000000010110000
PS C:\Users\Fabi\Documents\U\GitHub\Correccion_Deteccion>
```

- ¿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, demuéstrelo con su implementación)

Sí, es posible manipular bits para que Fletcher no detecte el cambio. Esto se debe a que el algoritmo sólo verifica que dos sumas acumuladas sean iguales módulo un número fijo. Si el error está diseñado para que esas dos sumas no cambien (colisión), el checksum resultará idéntico, aunque los datos estén dañados.

- Básicamente dos mensajes en binario pueden resultar en el mismo checksum y no hay manera de comprobar que se modificó.
- Por ejemplo, el cambiar el orden de los bytes puede llevar a este error.

Este es un ejemplo

Usando Fletcher-16

Mensaje de prueba: 11111111 00000000

Resultado: 11111111000000000000001100000001

Modificado: 00000000111111110000001100000001

Resultado: El receptor no detecta error



```
Bienvenido. Esta es una simulación de Checksum Fletcher
Elige el tipo con el que quieres trabajar. (Escribe 1, 2 o 3)
1. Fletcher-8
2. Fletcher-16
3. Fletcher-32
2

Genial, estamos usando Fletcher-16.
Ahora escribe el mensaje que quieres enviar (i.e.: 110101):
1111111100000000

El resultado de checksum es:
0000001100000001

El mensaje para el receptor:
11111111000000000000001100000001
PS C:\Users\Fabi\Documents\UAGitHub\Correccion_Deteccion>
```

```
Bienvenido. Esta es una simulación del RECEPTOR de Checksum Fletcher
Elige el tipo con el que quieres trabajar. (Escribe 1, 2 o 3)
1. Fletcher-8
2. Fletcher-16
3. Fletcher-32
2

Genial, estamos usando Fletcher-16.
Ahora la respuesta del emisor (i.e.: 110101):
000000011111110000001100000001

No se detectaron errores.
El mensaje es 0000000111111111
PS C:\Users\Fabi\Documents\UAGitHub\Correccion_Deteccion>
```

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

Comparando este algoritmo con códigos de Hamming (el otro método programado) y CRC-32 (otro método de detección), encontramos las siguientes ventajas:

- Primeramente, Fletcher-checksum es mucho más rápido de implementar y ejecutar, lo que lo hace ideal para dispositivos con recursos limitados.
- Dada la posibilidad de elegir entre Fletcher-8, 16 o 32 según el balance entre detección y tamaño extra, el overhead se reduce
- En general, su código es más simple y fácil de depurar.

Definitivamente hay desventajas, estas son algunas:

- No corrige errores como Hamming.
- Según lo investigado de CRC-32 y lo experimentado acá, la capacidad de detección de Fletcher-checksum es menor que CRC-32, especialmente en errores de ráfaga largos o patrones repetitivos.
- Como vimos en la pregunta anterior, hay posibilidad de errores por lo que no es ideal para entornos con mucho ruido o alta probabilidad de errores múltiples.

III. Implementación de la parte 2

a. Descripción general

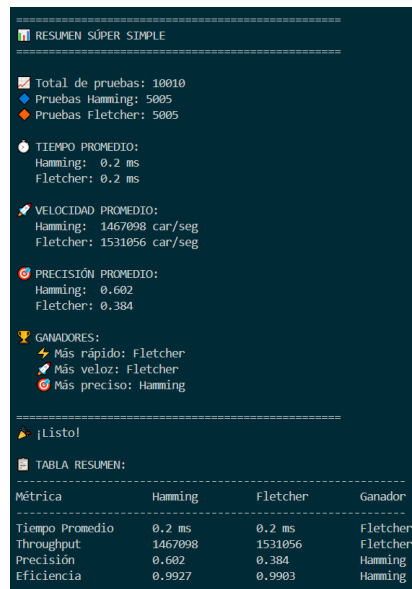
Utilizamos un protocolo TCP para la transmisión de los mensajes, con sockets, entre emisor y receptor. Según lo establecido desde la parte 1, trabajamos el emisor con java y el receptor con python. En general, ambos ‘trozos’ de código tienen la arquitectura del sistema, interactuando con el usuario para recibir información y mostrarle los resultados obtenidos.

En el lado del emisor (Java), el programa solicita al usuario el algoritmo con el que desea trabajar (Códigos de Hamming o Fletcher checksum), así como el mensaje a transmitir. Si el mensaje no está en formato binario, este se convierte a su representación ASCII binaria en la capa de Presentación. Luego, en la capa de Enlace, se calcula la información de integridad de acuerdo con el esquema

elegido utilizando la implementación desarrollada en la parte 1. Antes de enviar, se aplica la capa de Ruido, que introduce errores aleatorios en la trama con una probabilidad definida. Finalmente, en la capa de Transmisión, el mensaje y los parámetros asociados se envían mediante un socket TCP al receptor, empaquetados en formato JSON para facilitar su interpretación.

En el lado del receptor (Python), el programa funciona como un servidor TCP, escuchando permanentemente en el puerto configurado. Una vez recibe la conexión y el mensaje JSON, lo interpreta y extrae los parámetros enviados por el emisor. En la capa de Enlace, verifica la integridad del mensaje usando el algoritmo correspondiente (desde la parte 1). Si no se detectan errores, la capa de Presentación decodifica el binario ASCII para reconstruir el mensaje original y mostrarlo al usuario. Si se detecta un error en Fletcher, el mensaje se descarta; si se tratara de Hamming, se intentaría corregirlo y se anuncia al usuario. Por último, el receptor envía una respuesta de confirmación al emisor antes de cerrar la conexión.

b. Pruebas y Resultados



```
=====
RESUMEN SÚPER SIMPLE
=====
Total de pruebas: 10010
  Pruebas Hamming: 5005
  Pruebas Fletcher: 5005

TIEMPO PROMEDIO:
  Hamming: 0.2 ms
  Fletcher: 0.2 ms

VELOCIDAD PROMEDIO:
  Hamming: 1467098 car/seg
  Fletcher: 1531056 car/seg

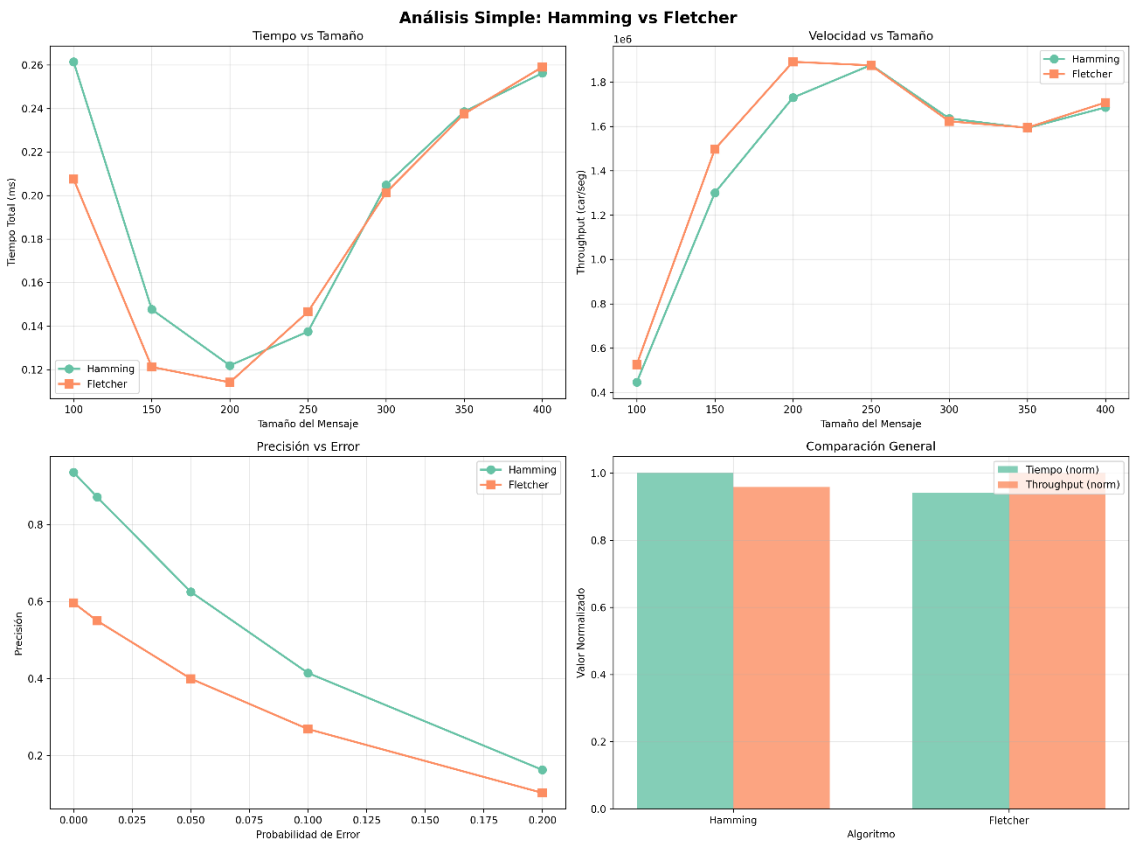
PRECISIÓN PROMEDIO:
  Hamming: 0.602
  Fletcher: 0.384

GANADORES:
  ⚡ Más rápido: Fletcher
  ⚡ Más veloz: Fletcher
  🎯 Más preciso: Hamming

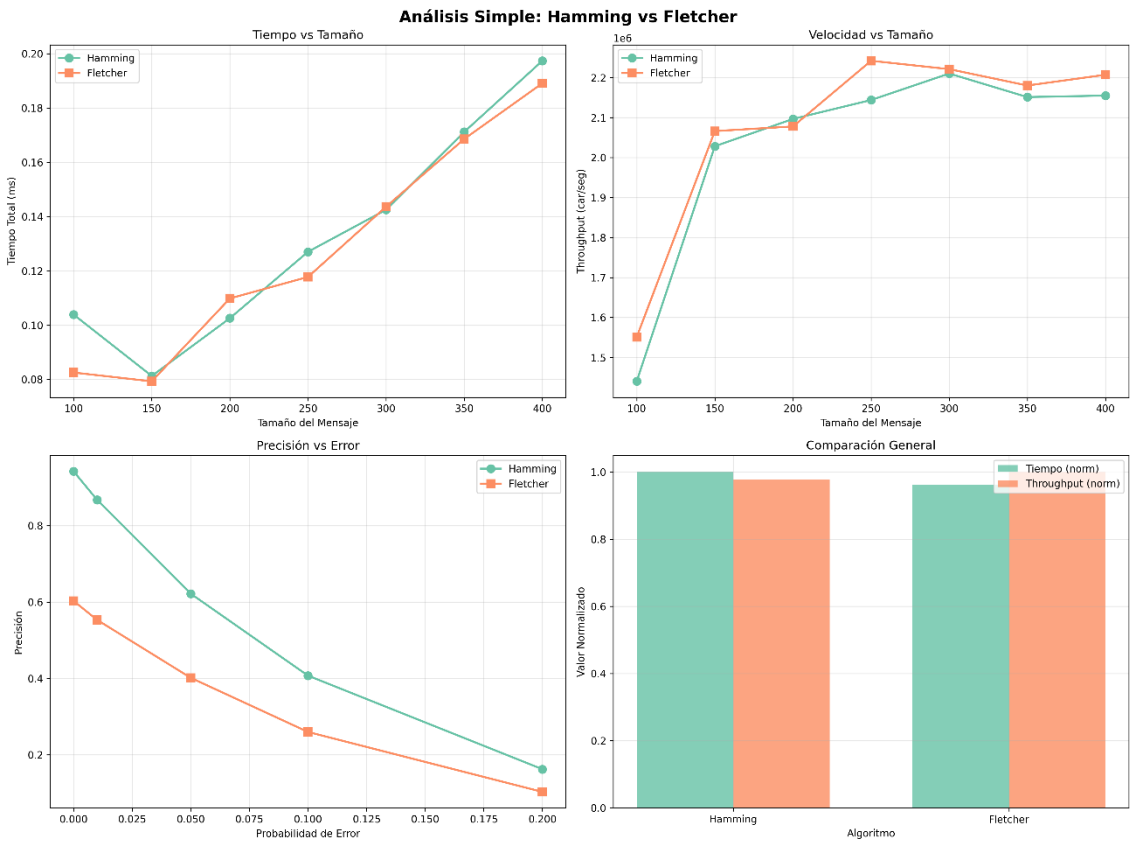
=====
¡Listo!
=====
TABLA RESUMEN:
=====
```

Métrica	Hamming	Fletcher	Ganador
Tiempo Promedio	0.2 ms	0.2 ms	Fletcher
Throughput	1467098	1531056	Fletcher
Precisión	0.602	0.384	Hamming
Eficiencia	0.9927	0.9903	Hamming

10K:



100K:



IV. GitHub

Link: https://github.com/Fabiola-cc/Correccion_Deteccion

V. Discusión

El análisis de rendimiento revela que ningún algoritmo es superior, cada uno destacando en contextos específicos según las prioridades del sistema. Hamming demuestra mejor funcionamiento para mensajes pequeños y cuando la integridad es crítica, manteniendo ~62% de precisión con 5% de error versus ~40% de Fletcher, debido a su capacidad de corrección automática. Fletcher sobresale en throughput alcanzando $\sim 1.9 \times 10^6$ caracteres/segundo y es más eficiente computacionalmente, siendo ideal cuando la velocidad de procesamiento es prioritaria sobre la recuperación de errores.

Hamming presenta significativamente mayor flexibilidad para manejar tasas de error elevadas gracias a su capacidad dual de detectar hasta 2 errores y corregir 1 automáticamente. Esta característica permite recuperar mensajes con errores simples sin retransmisión, mostrando degradación gradual de precisión (~40% con 10% de error) comparado con la caída abrupta de Fletcher (~25% con la misma tasa). Fletcher solo detecta errores sin corregirlos, descartando inmediatamente cualquier mensaje con checksum incorrecto, lo que lo hace menos robusto en entornos ruidosos pero más rápido en canales limpios.

La elección entre detección y corrección debe basarse en el balance entre confiabilidad del canal, costo de retransmisión y criticidad de los datos. Fletcher es óptimo para canales confiables (<1% error), recursos limitados y alta frecuencia de transmisión donde la retransmisión es rápida y económica, como redes Ethernet locales. Hamming es preferible para canales poco confiables (comunicaciones inalámbricas/satelitales), cuando la retransmisión es costosa o imposible, y para datos críticos en sistemas médicos o financieros donde la integridad es fundamental.

VI. Conclusiones

- Hamming supera a Fletcher en precisión manteniendo 62% vs 40% con 5% de probabilidad de error.
- Fletcher alcanza mayor throughput ($\sim 1.9 \times 10^6$ car/seg) siendo más eficiente para procesamiento de alta velocidad.
- Hamming es más robusto en entornos ruidosos al corregir errores automáticamente sin retransmisión.
- Fletcher es ideal para canales confiables donde la velocidad prima sobre la recuperación de errores.
- La elección depende del contexto: Hamming para integridad crítica, Fletcher para eficiencia computacional.

VII. Referencias

- Invarato, R. (2016, November 9). *Hamming* - Jarroba. Jarroba. <https://jarroba.com/hamming/>
- Bechtold, S. (2016). JUnit 5 User Guide. Junit.org. <https://docs.junit.org/current/user-guide/>