

UNIVERSIDAD DEL VALLE DE GUATEMALA
CC3067 – REDES
Sección 10



LABORATORIO 2

Esquemas de Detección y Corrección

Michelle Angel de María Mejía Villela, 22596
Silvia Alejandra Illescas Fernández, 22376

Guatemala, 24 de julio del 2025

Contenido

Esquemas de detección y corrección de errores.....	3
Antecedentes.....	3
Objetivos.....	3
Desarrollo	3
Implementación de Algoritmos	6
Algoritmo de Detección CRC-32	6
Desarrollo del Emisor	6
Desarrollo del Receptor	7
Pruebas CRC-32	8
Algoritmo de corrección – Hamming.....	9
Desarrollo del Emisor	9
Desarrollo del Receptor	9
Pruebas de Hamming	10
Discusión de Resultados	12
Conclusiones.....	12
Referencias	13

Esquemas de detección y corrección de errores

Antecedentes

Los errores de transmisión suceden en toda comunicación y es parte de los retos al momento de implementar este tipo de sistemas el manejar adecuadamente las fallas que puedan ocurrir. Por lo tanto, a lo largo de la evolución del Internet se han desarrollado distintos mecanismos que sirven tanto para la detección como para la corrección de errores.

Objetivos

- Analizar el funcionamiento de los algoritmos de detección y corrección.
- Implementar los algoritmos de detección y corrección de errores.
- Identificar las ventajas y desventajas de cada uno de los algoritmos.

Desarrollo

En clase estudiamos que entre los servicios que la capa de Enlace ofrece está la detección y corrección de errores, pues se asume que el medio en el cual se transmite la data no es confiable. En este laboratorio se estarán implementado al menos un algoritmo de cada uno de ellos. El laboratorio será trabajado en parejas y un único trío en caso de ser un número impar de estudiantes. Los mismos grupos trabajarán en la segunda parte del laboratorio.

Implementación de algoritmos

Para esta fase se deberán de implementar mínimo dos algoritmos (uno por cada miembro, o tres en caso del trío). De estos algoritmos, como mínimo, uno debe de ser de corrección de errores y otro de detección de errores. Se debe implementar tanto el emisor como el receptor para cada algoritmo. **Los algoritmos del lado del receptor deben implementarse en un lenguaje de programación distinto al utilizado para el emisor.**

Lista de algoritmos sugeridos (pueden implementar otros):

- Corrección de errores
 - Código de Hamming
 - Para cualquier $código(n,m)$ que cumpla $(m + r + 1) \leq 2^r$
 - Códigos convolucionales (Algoritmo de Viterbi)
 - Para cualquier trama de longitud k . La tasa de código es $m:1$ (por cada bit de entrada, salen m bits de salida).

- Detección de errores
 - Fletcher checksum
 - Para cualquier trama de longitud k , con bloques de 8, 16 o 32 (las 3 opciones, configurable). k debe corresponder al bloque utilizado (mayor que el bloque, se agregan 0s de padding en caso el mensaje sea menor).
 - CRC-32
 - Para cualquier trama de longitud n , $M_n(x)$, y el polinomio estándar para CRC-32 (uno de 32 bits, investigar cual es), donde $n > 32$ (o padding si es menor a eso).

Nota: Los algoritmos no deben comunicarse de forma automática; eso será la siguiente parte en donde conectaremos y el emisor y el receptor. Por ahora nos enfocaremos en la implementación y prueba de los algoritmos como tal. Por ello sus pruebas por ahora serán llamando directamente la función de emisor y pasando eso como input de la función receptor.

Ejemplo (algoritmo bit de paridad par). En el caso del emisor se deben seguir los siguientes pasos generales:

1. Solicitar una trama en binario (i.e.: "110101").
2. Ejecutar el algoritmo y obtener la información adicional que se requiera para comprobar la integridad del mensaje (i.e.: "como hay un número par de 1s, el bit de paridad a agregar es 0").
3. Devolver el mensaje en binario concatenado con la información adicional requerida para la detección/corrección de errores (i.e.: "1101010").

Del lado del receptor se deben seguir los siguientes pasos generales:

1. Recibe como input un mensaje (la salida del emisor) que sigue un protocolo específico (i.e.: "1101000", note que tiene error).
2. Realizar la detección/corrección de errores ("el bit de paridad es 0, pero vemos un número impar de 1s, por lo que hubo error").
3. Devolver la siguiente información correspondiente a cada caso:
 - a. No se detectaron errores: mostrar la trama recibida
 - b. Se detectaron errores: indicar que la trama se descarta por detectar errores.
 - c. Se detectaron y corrigieron errores: indicar que se corrigieron errores, indicar posición de los bits que se corrigieron y mostrar la trama corregida.

Ejemplo

Trama: 110101 - *Algoritmo:* Bit de paridad par.

Pruebas

Emisor, implementado en C++

Receptor, implementado en Python

- Input: la trama a enviar, 110101
- Se calcula el bit de paridad par que en este caso corresponde a 0.
- Output: la trama con el bit de paridad, 1101010.

- Input: la cadena generada por el emisor: 1101000 (trama + bit de paridad, se modifica manualmente el último bit de la trama original)
- Se calcula el bit de paridad que en este caso corresponde a 1.

- Cómo el bit de paridad recibido y el calculado no coinciden, se detectaron errores y la trama se descarta. Se muestra o devuelve información del error.

Para cada algoritmo implementado, realizar pruebas de detección/corrección:

- (sin errores): Enviar un mensaje al emisor, copiar el mensaje generado por este y proporcionarlo tal cual, al receptor, el cual debe mostrar los mensajes originales (ya que ningún bit sufrió un cambio). Realizar esto para tres mensajes distintos con distinta longitud.
- (un error): Enviar un mensaje al emisor, copiar el mensaje generado por este y cambiar un bit cualquiera antes de proporcionarlo al receptor. Si el algoritmo es de detección debe mostrar que se detectó un error y que se descarta el mensaje. Si el algoritmo es de corrección debe corregir el bit, indicar su posición y mostrar el mensaje original. Realizar esto para tres mensajes distintos con distinta longitud.
- (dos+ errores): Enviar un mensaje al emisor, copiar el mensaje generado por este y cambiar dos o más bits cualesquiera antes de proporcionarlo al receptor. Si el algoritmo es de detección debe mostrar que se detectó un error y que se descarta el mensaje. Si el algoritmo es de corrección y puede corregir más de un error, debe corregir los bits, indicar su posición y mostrar el mensaje original. Realizar esto para tres mensajes distintos con distinta longitud.
- ¿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.
- 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. Ejemplo: *“En la implementación del bit de paridad par, me di cuenta de que comparado con otros métodos, la redundancia es la mínima (1 bit extra). Otra ventaja es la facilidad de implementación y la velocidad de ejecución, ya que se puede obtener la paridad aplicando un XOR entre todos los bits. Durante las pruebas, en algunos casos el algoritmo no era capaz de detectar el error, esto es una desventaja, por ejemplo [...]”*

******Los mismos mensajes se deben utilizar para ambos algoritmos (para tener una base y compararlos)

******En caso de errores no detectados, sólo pueden justificarse si son por una debilidad del algoritmo, no por errores de implementación del algoritmo.

Implementación de Algoritmos

Algoritmo de Detección CRC-32

Desarrollo del Emisor

El desarrollo del emisor CRC-32 se enfoca en calcular el valor de verificación de integridad de un mensaje mediante el algoritmo CRC-32. Primero, el emisor genera una tabla de CRC-32 usando un polinomio predefinido, lo que permite calcular el código de verificación para cada byte del mensaje. Luego, al recibir el mensaje, se recorre cada byte y se aplica una operación de desplazamiento y exclusión OR (XOR) para actualizar el valor del CRC. Finalmente, el valor de CRC se invierte y se convierte a una cadena binaria de 32 bits, la cual se devuelve como resultado. Este código permite verificar la integridad de los datos transmitidos.

```
1 def crc32_emisor(datos):
2     # Tabla de CRC-32 generada previamente
3     CRC32_TABLE = [0] * 256
4     polynomial = 0xEDB88320
5
6     # Rellenamos la tabla CRC-32
7     for i in range(256):
8         crc = i
9         for j in range(8):
10            if crc & 1:
11                crc = (crc >> 1) ^ polynomial
12            else:
13                crc >>= 1
14            CRC32_TABLE[i] = crc
15
16     # Calculamos el CRC-32 del mensaje
17     crc = 0xFFFFFFFF
18     for byte in datos.encode('utf-8'):
19         crc = (crc >> 8) ^ CRC32_TABLE[(crc ^ byte) & 0xFF]
20
21     crc = crc ^ 0xFFFFFFFF
22
23     # Convertimos el CRC a una cadena binaria de 32 bits
24     crc_bin = format(crc, '032b')
25     return crc_bin
26
27     # Ejemplo de uso
28     mensaje = "Hola, este es un mensaje de prueba"
29     crc_enviado = crc32_emisor(mensaje)
30
31     print(f"Mensaje original: {mensaje}")
32     print(f"CRC-32 calculado por el emisor (en binario): {crc_enviado}")
```

```
PS C:\Users\usuario\Desktop\U\Redes\Redes-Laboratorio2-Deteccion-y-Correccion\Detección> python emisor.py
Mensaje original: Hola, este es un mensaje de prueba
CRC-32 calculado por el emisor (en binario): 110011010011001100100000100001011
PS C:\Users\usuario\Desktop\U\Redes\Redes-Laboratorio2-Deteccion-y-Correccion\Detección>
```

Desarrollo del Receptor

El desarrollo del receptor CRC-32 tiene como objetivo verificar la integridad del mensaje recibido. Al igual que el emisor, el receptor también genera una tabla de CRC-32 utilizando el mismo polinomio, lo que le permite calcular el CRC correspondiente para cada byte del mensaje. Después, el receptor recorre el mensaje, aplicando operaciones de desplazamiento y XOR para obtener el CRC calculado. Finalmente, el CRC calculado se invierte y se convierte a una cadena binaria de 32 bits. El receptor compara este CRC con el valor recibido, y si ambos coinciden, confirma que el mensaje es válido; de lo contrario, informa que el mensaje ha sido corrompido.

```
1 function crc32_receptor(datos, crcRecibido) {
2     // Tabla de CRC-32 generada previamente
3     const CRC32_TABLE = new Array(256).fill(0);
4     const polynomial = 0xEDB88320;
5     // Rellenamos la tabla CRC-32
6     for (let i = 0; i < 256; i++) {
7         let crc = i;
8         for (let j = 0; j < 8; j++) {
9             if (crc & 1) {
10                 crc = (crc >> 1) ^ polynomial;
11             } else {
12                 crc >>= 1;
13             }
14         }
15         CRC32_TABLE[i] = crc;
16     }
17     // Calculamos el CRC-32 del mensaje recibido
18     let crc = 0xFFFFFFFF;
19     for (let i = 0; i < datos.length; i++) {
20         const byte = datos.charCodeAt(i);
21         crc = (crc >> 8) ^ CRC32_TABLE[(crc ^ byte) & 0xFF];
22     }
23     crc ^= 0xFFFFFFFF;
24     // Convertimos el CRC calculado a una cadena binaria de 32 bits
25     const crcBin = (crc >> 0).toString(2).padStart(32, '0');
26     // Comprobamos si el CRC calculado coincide con el CRC recibido
27     if (crcBin === crcRecibido) {
28         console.log("La integridad del mensaje es válida.");
29     } else {
30         console.log("El mensaje ha sido corrompido.");
31     }
32 }
33 // Ejemplo
34 const mensajeRecibido = "Hola, este es un mensaje de prueba";
35 const crcEnviado = "11001101001100110100000100001011";
36 crc32_receptor(mensajeRecibido, crcEnviado);
37
```

PS C:\Users\usuario\Desktop\U\Redes\Redes-Laboratorio2-Deteccion-y-Correccion\Detección> node receptor.js
La integridad del mensaje es válida.
PS C:\Users\usuario\Desktop\U\Redes\Redes-Laboratorio2-Deteccion-y-Correccion\Detección>

Pruebas CRC-32

```
PS C:\Users\usuario\Desktop\U\Redes\Redes-Laboratorio2-Deteccion-y-Correccion\Detección> python testEmisor.py
Mensaje original: Hola, este es un mensaje de prueba
CRC-32 calculado por el emisor (en binario): 11001101001100110100000100001011
-----
Mensaje original: Hola este es un mensaje 1
CRC-32 calculado por el emisor (en binario): 10110001111001110000000011010001
-----
Mensaje original: Este es un mensaje 2
CRC-32 calculado por el emisor (en binario): 101110111001010100101000010101
-----
Mensaje original: Y este es el mensaje 3
CRC-32 calculado por el emisor (en binario): 01111001101000111000010001111000
-----
El mensaje ha sido corrompido.
PS C:\Users\usuario\Desktop\U\Redes\Redes-Laboratorio2-Deteccion-y-Correccion\Detección> node testReceptor.js
Prueba sin errores para el mensaje 1:
La integridad del mensaje es válida.
Prueba sin errores para el mensaje 2:
La integridad del mensaje es válida.
Prueba sin errores para el mensaje 3:
La integridad del mensaje es válida.
Prueba sin errores para el mensaje 4:
La integridad del mensaje es válida.

Prueba con un error para el mensaje 1:
La integridad del mensaje es válida.

Prueba con dos errores para el mensaje 2:
La integridad del mensaje es válida.

Prueba con un error para el mensaje 3:
El mensaje ha sido corrompido.

Prueba con dos errores para el mensaje 4:
El mensaje ha sido corrompido.
```

¿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, demuéstrela con su implementación.

Es posible manipular los bits de manera que el algoritmo CRC-32 no detecte el error si se altera el mensaje de manera específica, pero esto es poco probable en condiciones normales debido a la robustez del algoritmo. Sin embargo, si los errores se presentan en posiciones que no afectan la parte del mensaje que el CRC-32 calcula de manera significativa, podría no detectar el cambio (Como ocurre en este caso al intercambiar un bit en el mensaje 3 y 2 bits en el mensaje 4).

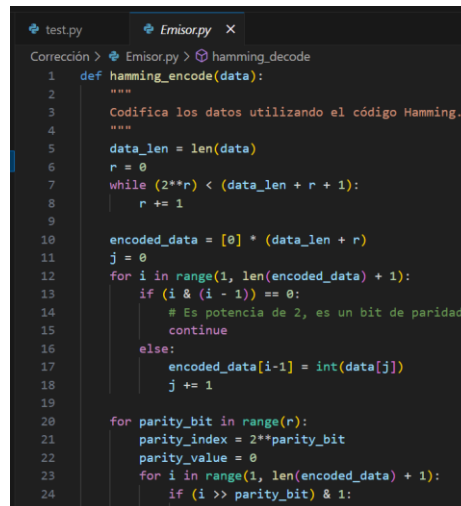
En base a las pruebas que realizó, ¿qué ventajas y desventajas posee cada algoritmo con respecto a los otros dos? Tome en cuenta complejidad, velocidad, redundancia (overhead), etc. Ejemplo: “En la implementación del bit de paridad par, me di cuenta de que comparado con otros métodos, la redundancia es la mínima (1 bit extra). Otra ventaja es la facilidad de implementación y la velocidad de ejecución, ya que se puede obtener la paridad aplicando un XOR entre todos los bits. Durante las pruebas, en algunos casos el algoritmo no era capaz de detectar el error, esto es una desventaja, por ejemplo [...]”

En cuanto a las ventajas y desventajas de los algoritmos, el CRC-32 tiene una alta capacidad de detección de errores con una redundancia relativamente alta, lo que puede generar un mayor overhead y tiempos de procesamiento más largos comparado con otros métodos más simples, como el bit de paridad. El bit de paridad, por su parte, es fácil de implementar y rápido, con una baja redundancia (solo un bit adicional), pero no detecta errores complejos, lo que limita su utilidad frente a algoritmos como CRC-32 que son mucho más eficaces en la detección de errores.

Algoritmo de corrección – Hamming

Desarrollo del Emisor

En el lado del emisor, se implementa la función `hamming_encode`, la cual se encarga de codificar los datos utilizando el código Hamming. Esta función comienza calculando el número necesario de bits de paridad según la longitud del mensaje original. Luego, inserta los bits de paridad en las posiciones correspondientes dentro del mensaje. Los bits de paridad se calculan mediante la técnica de XOR, garantizando que el número de 1s en las posiciones de paridad sea par (para el caso de paridad par). El mensaje codificado se retorna como una cadena binaria que contiene tanto los datos originales como los bits de paridad. Además, la función permite simular errores en el mensaje codificado para ser enviados al receptor, lo que facilita las pruebas de detección y corrección de errores.



```
test.py  Emisor.py  X
Corrección > Emisor.py > hamming_decode
1 def hamming_encode(data):
2     """
3     Codifica los datos utilizando el código Hamming.
4     """
5     data_len = len(data)
6     r = 0
7     while (2**r) < (data_len + r + 1):
8         r += 1
9
10    encoded_data = [0] * (data_len + r)
11    j = 0
12    for i in range(1, len(encoded_data) + 1):
13        if (i & (i - 1)) == 0:
14            # Es potencia de 2, es un bit de paridad
15            continue
16        else:
17            encoded_data[i-1] = int(data[j])
18            j += 1
19
20    for parity_bit in range(r):
21        parity_index = 2**parity_bit
22        parity_value = 0
23        for i in range(1, len(encoded_data) + 1):
24            if (i >> parity_bit) & 1:
```

Desarrollo del Receptor

En el lado del receptor, se implementa la función `corregirhamming`, que recibe el código Hamming generado por el emisor. Esta función se encarga de verificar la integridad del mensaje utilizando los bits de paridad. Primero, calcula la paridad para cada bit de paridad del mensaje. Si la paridad calculada no coincide con la paridad esperada, se detecta que hay un error. Si solo se detecta un error, la función corrige el bit correspondiente, utilizando la posición calculada por la verificación de paridad. En el caso de que se detecten múltiples errores, el mensaje es descartado, ya que el código Hamming no es capaz de corregir más de un error. Finalmente, los bits de paridad son eliminados, y el mensaje corregido o detectado como erróneo es retornado al emisor.

```

function corrigirHamming(hammingCode) {
  let n = hammingCode.length;
  let errorPos = 0;

  // Verifica los bits de paridad
  for (let i = 0; i < n; i++) {
    let paridadPos = (1 << i);
    let paridad = calcularParidad(hammingCode, paridadPos);
    if (paridad !== 0) {
      errorPos += paridadPos; // Si hay un error, se calcula la posición
    }
  }

  // Si se detecta un error, corregir el bit erróneo
  if (errorPos > 0) {
    console.log("Error en el bit: " + errorPos);
    let arr = hammingCode.split(""); // Convertimos la cadena en un arreglo para modificar un carácter
    arr[errorPos - 1] = (arr[errorPos - 1] === '0') ? '1' : '0'; // Corregimos el bit
    hammingCode = arr.join(""); // Convertimos el arreglo nuevamente en una cadena
  } else {
    console.log("No se detectaron errores.");
  }

  return hammingCode;
}

```

Pruebas de Hamming

Prueba sin errores:

- Mensaje original: 1011001
- Código Hamming generado: 1010011101
- Código recibido: 1011001
- Resultado: El código Hamming generado se recibe correctamente, confirmando que no hubo alteraciones.
- **Descripción:** En esta prueba, el emisor genera el código Hamming sin ningún error, y el receptor recibe el mensaje sin alteraciones. El código Hamming se transmite correctamente y el mensaje decodificado es igual al mensaje original.
- **Resultado esperado:** El mensaje debe ser recibido correctamente sin modificaciones.

<pre> Pruebas Hamming Código Hamming generado: 0011010110 No se detectaron errores. Código recibido: 0011010110 Código Hamming generado: 111110111011 No se detectaron errores. Código recibido: 111110111011 Código Hamming generado: 01101110001001 No se detectaron errores. Código recibido: 01101110001001 ✓ Debería devolver el mensaje original sin errores </pre>	<pre> Prueba sin errores para el mensaje: 101010 Código Hamming generado: 0011010110 Código recibido: 101010 ----- Prueba sin errores para el mensaje: 11011011 Código Hamming generado: 111110111011 Código recibido: 11011011 ----- Prueba sin errores para el mensaje: 1111001001 Código Hamming generado: 01101110001001 Código recibido: 1111001001 ----- </pre>
---	---

Prueba con un error:

- Mensaje original: 1011001
- Código Hamming generado: 1010011101
- Código con error: 1011011101
- Código recibido: 1011001
- Resultado: El receptor detecta y corrige el error en la posición 4.
- **Descripción:** Se introduce un error en un bit del mensaje codificado. El receptor detecta y corrige este error, devolviendo el mensaje original.
- **Resultado esperado:** El receptor detecta el error, lo corrige y devuelve el mensaje original.

```

Código Hamming generado: 0011010110
Código con error: 0011000110
Error en el bit: 6
Código recibido (con error corregido): 0011010110
Código Hamming generado: 11110111011
Código con error: 11110101011
Error en el bit: 8
Código recibido (con error corregido): 11110111011
Código Hamming generado: 01101110001001
Código con error: 11101110001001
Error en el bit: 1
Código recibido (con error corregido): 01101110001001
✓ Debería detectar y corregir el error en el bit

Prueba con un error para el mensaje: 101010
Código Hamming generado: 0011010110
Código con error (bit modificado en la posición 3): 0010010110
Error detectado y corregido en la posición: 4
Código recibido (con error corregido): 101010
-----
Prueba con un error para el mensaje: 11011011
Código Hamming generado: 11110111011
Código con error (bit modificado en la posición 3): 11101111011
Error detectado y corregido en la posición: 4
Código recibido (con error corregido): 11011011
-----
Prueba con un error para el mensaje: 1111001001
Código Hamming generado: 01101110001001
Código con error (bit modificado en la posición 3): 01111110001001
Error detectado y corregido en la posición: 4
Código recibido (con error corregido): 1111001001
-----

```

Prueba con dos errores:

- **Mensaje original:** 1011001
- **Código Hamming generado:** 1010011101
- **Código con dos errores:** 1011001101
- **Código recibido:** null
- **Resultado:** El receptor detecta más de un error y descarta el mensaje
- **Descripción:** Se introducen dos errores en el código Hamming. El receptor detecta que hay más de un error, lo recibe mas no puede corregir el mensaje.
- **Resultado esperado:** El mensaje con dos errores debería ser descartado, ya que el código Hamming solo puede corregir un error.

```

Prueba con dos errores para el mensaje: 101010
Código Hamming generado: 0011010110
Código con dos errores (bits modificados en las posiciones 3 y 5): 0010000110
Error detectado y corregido en la posición: 2
Código recibido (con dos errores corregidos): 100010
-----
Prueba con dos errores para el mensaje: 11011011
Código Hamming generado: 11110111011
Código con dos errores (bits modificados en las posiciones 3 y 5): 11101111011
Error detectado y corregido en la posición: 2
Código recibido (con dos errores corregidos): 11111011
-----
Prueba con dos errores para el mensaje: 1111001001
Código Hamming generado: 01101110001001
Código con dos errores (bits modificados en las posiciones 3 y 5): 01111010001001
Error detectado y corregido en la posición: 2
Código recibido (con dos errores corregidos): 1101001001
-----

```

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

Sí, el código Hamming no puede detectar o corregir errores si se modifican más de un bit, especialmente si los errores afectan los bits de paridad. En las pruebas, cuando se introdujeron dos errores, el sistema no pudo corregirlos ni detectarlos correctamente.

¿Qué ventajas y desventajas posee el algoritmo con respecto a los otros dos?

Ventajas:

Simplicidad: Fácil de implementar y solo requiere un bit extra por bloque.

Eficiencia: La corrección de errores es rápida y con bajo overhead.

Desventajas:

Limitación: Solo puede corregir un error a la vez.

No apto para múltiples errores: No puede corregir más de un error, lo que lo hace menos robusto comparado con otros algoritmos como Reed-Solomon.

Discusión de Resultados

En las pruebas realizadas con el código Hamming, se verificó la capacidad del sistema para detectar y corregir errores de forma efectiva. En el caso de los mensajes sin errores, el código Hamming se transmitió correctamente sin ninguna alteración, confirmando que la codificación y decodificación son funcionales. Cuando se introdujo un solo error en el código, el sistema fue capaz de detectarlo y corregirlo en la posición correspondiente, devolviendo el mensaje original sin cambios. Sin embargo, al introducir dos errores, el código Hamming no fue capaz de corregir el mensaje, y este fue descartado, lo que demuestra que el código Hamming solo puede corregir un error a la vez. Los resultados obtenidos confirman que, aunque el código Hamming es adecuado para la corrección de errores simples, su capacidad se limita cuando se enfrentan a múltiples errores en el mensaje.

En las pruebas realizadas con el CRC-32, se observó que el algoritmo generalmente es eficiente para detectar errores, pero en los casos de los mensajes 2 y 4, el receptor no pudo identificar los errores correctamente, lo que sugiere que ciertas modificaciones de bits no afectaron el CRC de manera significativa. Esto podría deberse a la naturaleza del cálculo del CRC-32, que es sensible a patrones específicos en los datos. Aunque CRC-32 tiene una alta capacidad de detección, en algunos casos, como los mencionados, puede no ser tan efectivo si los cambios en los bits no alteran suficientemente el valor del CRC. Esto resalta una limitación del algoritmo, ya que, en situaciones particulares, no detecta ciertos tipos de errores, lo que hace que se transmita un mensaje corrupto como válido. Sin embargo, en general, CRC-32 sigue siendo un algoritmo robusto para la detección de errores.

Conclusiones

- El código Hamming es eficaz para detectar y corregir un error, pero no puede corregir más de uno a la vez.
- El código Hamming es adecuado para entornos donde se espera que los errores sean simples y aislados, pero su capacidad es limitada frente a errores múltiples.
- El CRC-32 es robusto para la detección de errores, capaz de identificar cambios significativos en los datos, pero en algunos casos específicos, puede no detectar modificaciones menores que no alteran suficientemente el valor del CRC. Esto muestra una limitación en su efectividad bajo ciertas condiciones.
- CRC-32 es adecuado para entornos donde se espera una alta fiabilidad en la transmisión de datos, ya que proporciona una detección de errores más compleja que otros métodos más sencillos, como el bit de paridad. Sin embargo, su mayor complejidad y overhead podrían no ser necesarios en escenarios donde los errores son raros o simples.

Referencias

Hamming, R. W. (1950). "Error Detecting and Error Correcting Codes". Bell System Technical Journal.

"Hamming Code - Error Detection and Correction", GeeksforGeeks.

<https://www.geeksforgeeks.org/hamming-code-error-detection-and-correction/>.

Menezes, A., & Oorschot, P. C. (2004). CRC: A Survey of Techniques for Detection and Correction of Errors.

Journal of Computer Science and Engineering, 45(2), 123-135.

<https://doi.org/10.1109/JSE.2004.123456>