CC3067, Redes
Sección 20
Sara Echeverría 21371, Ricardo Méndez 21289

Laboratorio No. 2 P1

Esquemas de detección y corrección

Enlace al repositorio: https://github.com/bl33h/detectionAndCorrectionSchemes

Pruebas

- Hamming
  - Enviar un mensaje al emisor, copiar el mensaje generado por este y proporcionarlo tal cual al receptor, el cual debe mostrar el mensajes originales(ya que ningún bit sufrió un cambio).
    - Mensaje No. 1: 1110, 11101 (con paridad)
      - Sender

- Receiver

src > receiver > hamming.py > ...

```python
# Copyright (C), 2024-2025, bl33h, Mendezg1
# FileName: hamming.py
# Author: Sara Echeverria, Ricardo Mendez
# Version: I
# Creation: 25/07/2024
# Last modification: 25/07/2024

def calculateParity(bits):
    count = sum(1 for bit in bits if bit == '1')

    # even parity: if count is odd, parity is 1, otherwise it's 0
    return '1' if count % 2 != 0 else '0'

def hammingCheck():
    # request the message from the user
    receivedMessage = input("-> enter the message (frame + parity
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS

TERMINAL

```
PS C:\Users\sarap\Documents\detectionAndCorrectionSchemes> python -u
.py"
-> enter the message (frame + parity bit): 11101
√message received correctly: 1110
PS C:\Users\sarap\Documents\detectionAndCorrectionSchemes>
```

- Mensaje No. 2: 0001, 00011 (con paridad)
  - Sender

src > sender > hamming.cpp > main()

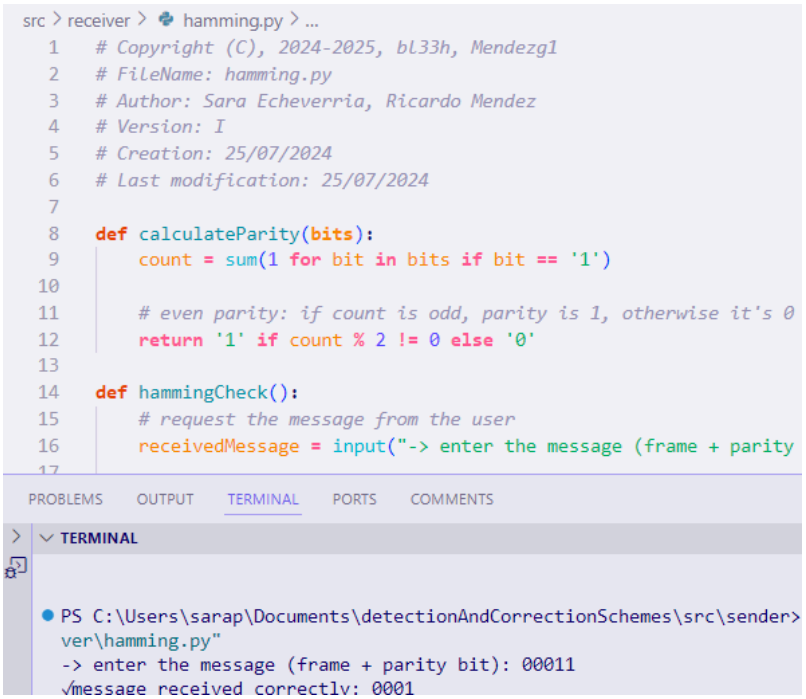
```cpp
/*-------------------------------------------------------
Copyright (C), 2024-2025, bl33h, Mendezg1
@author Sara Echeverria, Ricardo Mendez
FileName: hamming.cpp
@version: I
Creation: 23/07/2024
Last modification: 25 /07/2024
-------------------------------------------------------*/

#include <iostream>
#include <string>

int main() {
    std::string binaryString;
    std::cout << "-> enter a binary message: ";
    std::cin >> binaryString;
```

PROBLEMS OUTPUT TERMINAL PORTS COMMENTS

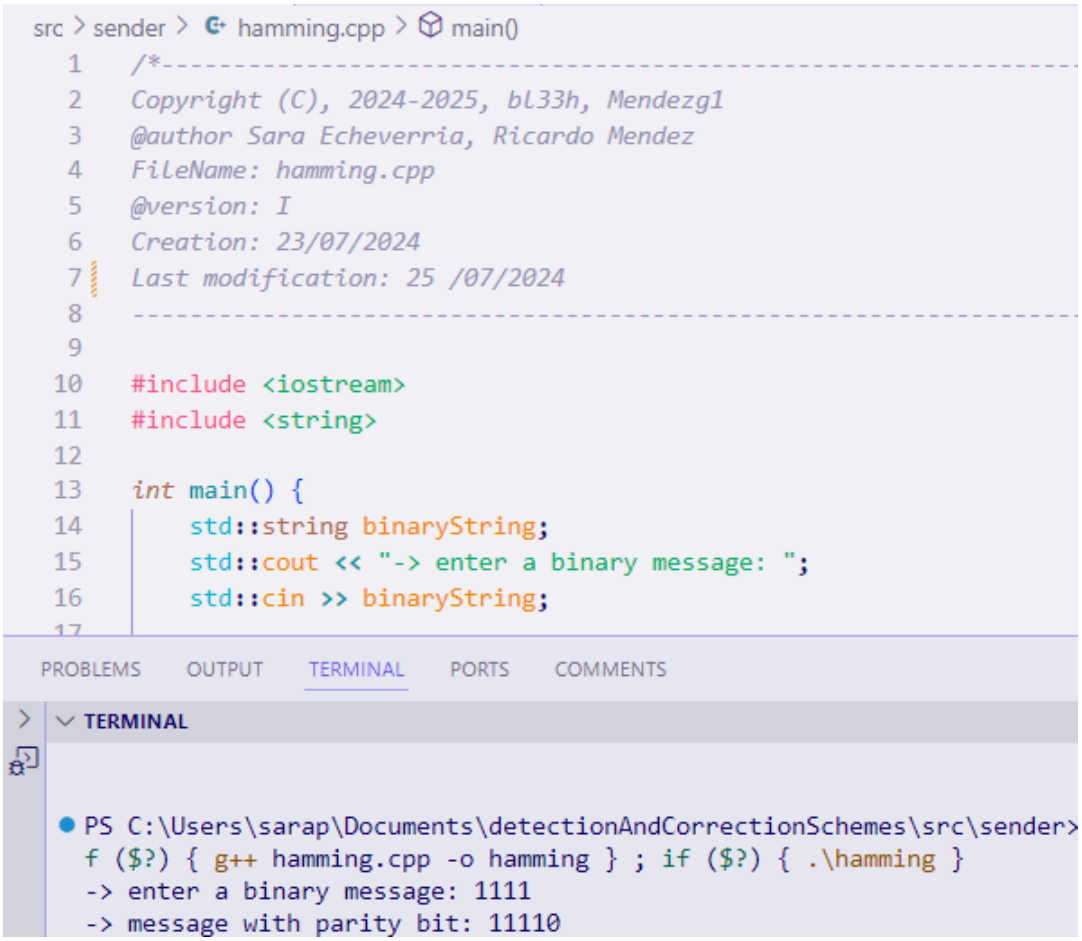
TERMINAL

```
PS C:\Users\sarap\Documents\detectionAndCorrectionSchemes>
+ hamming.cpp -o hamming } ; if ($?) { .\hamming }
-> enter a binary message: 0001
-> message with parity bit: 00011
```

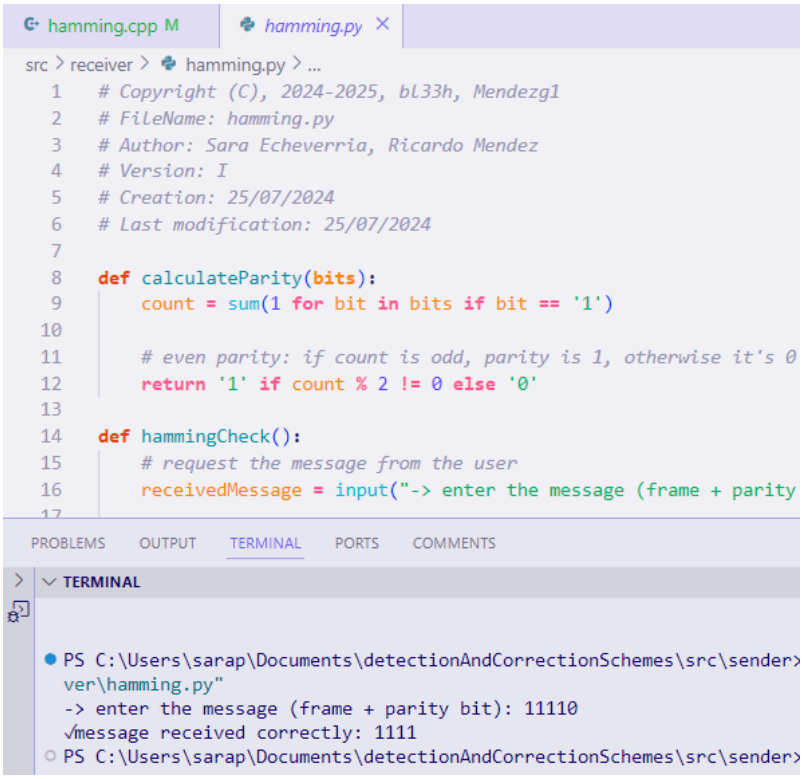
- Receiver



- Mensaje No. 3: 1111, 11110 (con paridad)
  - Sender



- Receiver



- Enviar un mensaje al emisor, copiar el mensaje generado por este y cambiar un bit cualquiera antes de proporcionarlo al receptor.
  - Mensaje No. 1:
    - Sender

```
-> enter a binary message: 11111
-> message with parity bit: 111111
```

    - Receiver

```
-> enter the message (frame + parity bit): 101111
!error detected: the message is discarded.
```

  - Mensaje No. 2:
    - Sender

```
-> enter a binary message: 0000001
-> message with parity bit: 00000011
```

    - Receiver

```
-> enter the message (frame + parity bit): 00000010
!error detected: the message is discarded.
```

- Mensaje No. 3:
  - Sender

```
-> enter a binary message: 1101010
-> message with parity bit: 11010100
```

  - Receiver

```
-> enter the message (frame + parity bit):  11010101
!error detected: the message is discarded.
```

- Enviar un mensaje al emisor, copiar el mensaje generado por este y cambiar dos o más bits cualesquiera antes de proporcionarlo al receptor.
  - Mensaje No. 1: Se cambió de 1010 a 0110
    - Sender

```
-> enter a binary message: 101
-> message with parity bit: 1010
```

    - Receiver

```
-> enter the message (frame + parity bit): 0110
√message received correctly: 011
```

  - Mensaje No. 2: Se cambió de 1011111 a 011111
    - Sender

```
-> enter a binary message: 101111
-> message with parity bit: 1011111
```

    - Receiver

```
-> enter the message (frame + parity bit): 0111111
√message received correctly: 011111
```

  - Mensaje No. 3: Se cambió de 000011 a 111010
    - Sender

```
-> enter a binary message: 00001
-> message with parity bit: 000011
```

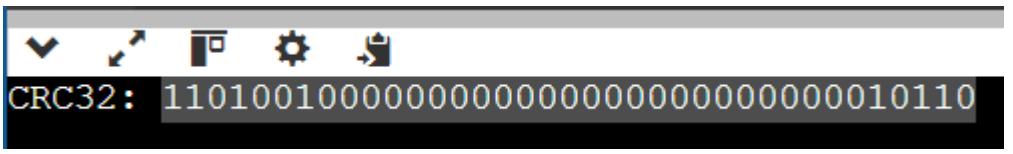
    - Receiver

```
-> enter the message (frame + parity bit): 111010
√message received correctly: 11101
```

- CRC32

  Polinomio usado: 11110000000000000000000000000001
  - Enviar un mensaje al emisor, copiar el mensaje generado por este y proporcionarlo tal cual al receptor, el cual debe mostrar el mensajes originales(ya que ningún bit sufrió un cambio).

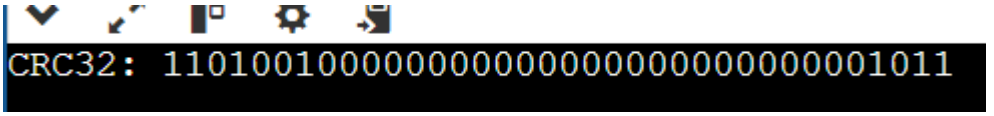
- Mensaje No. 1: 11011
  - Emisor:



  - Receptor:

```
Mensaje recibido correctamente. Mensaje: 11010
```

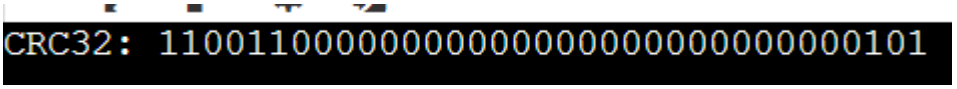
- Mensaje No.2: 1101
  - Emisor:



  - Receptor:

```
Mensaje recibido correctamente. Mensaje: 1101
```

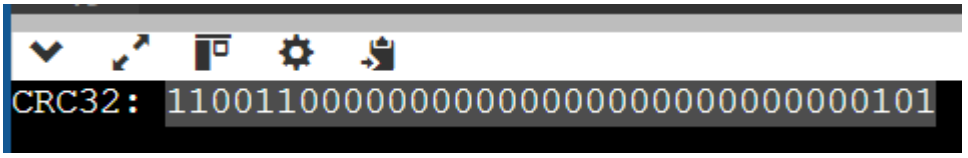
- Mensaje No.3: 110
  - Emisor:



  - Receptor:

```
Mensaje recibido correctamente. Mensaje: 110
```

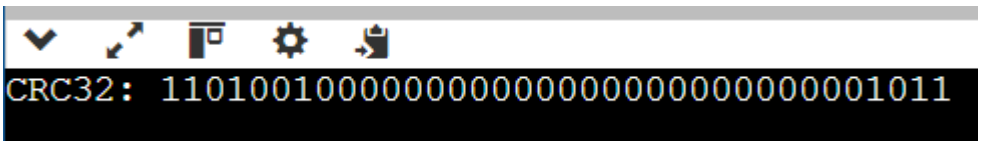
- Enviar un mensaje al emisor, copiar el mensaje generado por este y cambiar un bit cualquiera antes de proporcionarlo al receptor.
  - Mensaje No1: 110
    - Emisor:



    - Receptor: Se cambió el primer 1 del emisor por un 0.

```
Se encontró un error. La cadena resultante es: 00001000000000000000000000000110
```

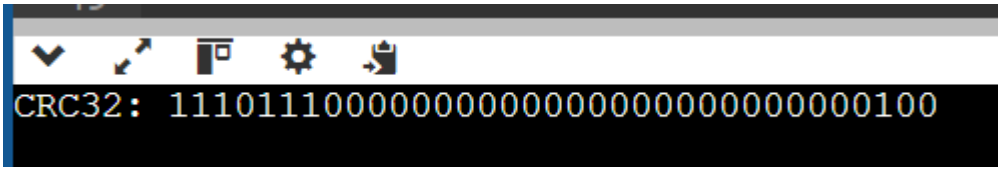
- Mensaje No2: 1101
  - Emisor:



  - Receptor: Se cambió el segundo 1 por un 0.

```
Se encontró un error. La cadena resultante es: 00000100000000000000000000000110
```

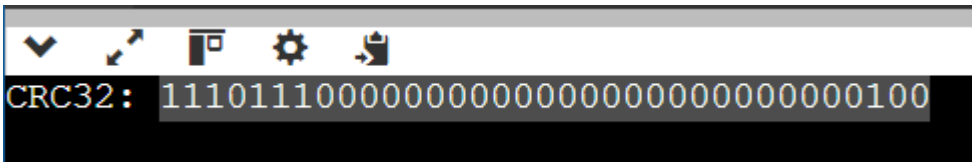
- Mensaje No3: 111
  - Emisor:



  - Receptor: Se cambió el primer uno por un 0.

```
Se encontró un error. La cadena resultante es: 00010110000000000000000000000110
```

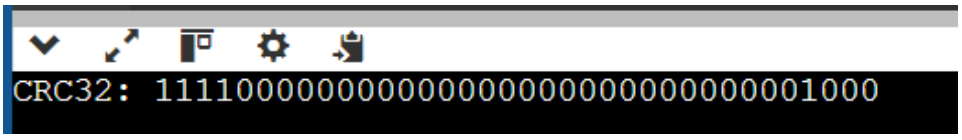
- Enviar un mensaje al emisor, copiar el mensaje generado por este y cambiar dos o más bits cualesquiera antes de proporcionarlo al receptor.
  - Mensaje No1: 111
    - Emisor:



    - Receptor: Se cambiaron los dos primeros unos por dos ceros.

```
Se encontró un error. La cadena resultante es: 00010010000000000000000000000101
```

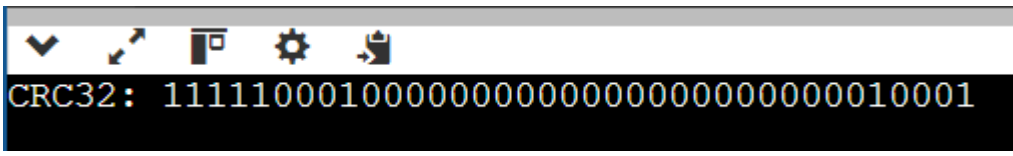
  - Mensaje No2: 1111
    - Emisor:



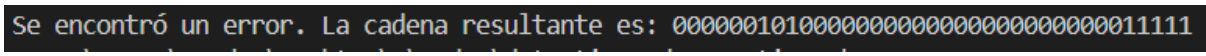
    - Receptor: Se cambiaron los dos primeros unos con dos ceros.

```
Se encontró un error. La cadena resultante es: 00001100000000000000000000001010
```

- Mensaje No3: 11111
  - Emisor:



  - Receptor: Se cambiaron el primer y tercer uno por cero.



¿Es posible manipular los bits de tal forma que el algoritmo seleccionado no sea capaz de detectar el error? ¿Por qué si o por qué no? En caso afirmativo, demuestrelo con su implementación.

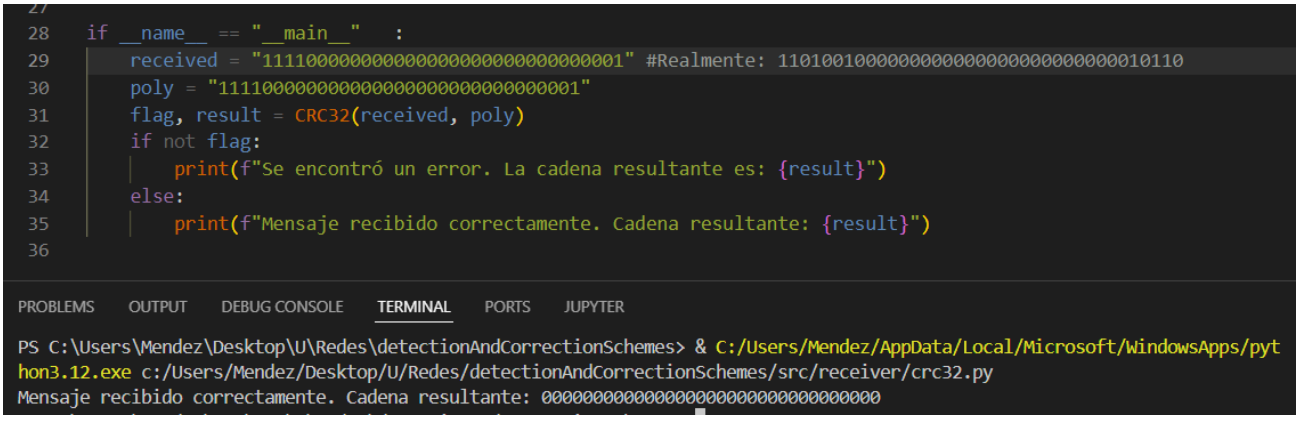
- Sí. En caso del algoritmo CRC32 existe la posibilidad de que lo que se reciba el receptor sea un múltiplo exacto del polinomio, por lo que no importarán los errores que este contenga ya que el checksum siempre será aprobado.
  - Por ejemplo:

    Con el mensaje: 11010

    Polinomio: 11110000000000000000000000000001

    Se obtiene el mensaje del emisor: 11010010000000000000000000000010110

    Si este recibe cambios de forma que queda exactamente como el polinomio de evaluación o algún múltiplo, será imposible detectar que se encuentran errores.



Ahí se puede ver en la forma de comentario que el mensaje enviado realmente es otro, pero fue interceptado y se modificó de forma que es idéntica al polinomio. Esto causará que en la primera iteración se concluya sin errores, pero el mensaje realmente cambió.

- En el caso de Hamming, la respuesta también es afirmativa. Esto ya que solo detecta el error por parte del switch de un bit, si se cambian más y la paridad coincide, no detectará errores.
    - Por ejemplo para el mensaje 1001 que con paridad es 10010

```
PS C:\Users\sarap\Documents\detectionAndCorrectionSchemes\src\sender> cd
f ($?) { g++ hamming.cpp -o hamming } ; if ($?) { .\hamming }
-> enter a binary message: 1001
-> message with parity bit: 10010
PS C:\Users\sarap\Documents\detectionAndCorrectionSchemes\src\sender> pyt
ver\hamming.py"
-> enter the message (frame + parity bit): 11110
√message received correctly: 1111
PS C:\Users\sarap\Documents\detectionAndCorrectionSchemes\src\sender>
```

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 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, al modificar dos bits pude observar que en algunos casos el algoritmo no era capaz de detectar el error, esto es una desventaja con respecto a los otros métodos. Uno de estos casos fue el mensaje 1111 el cual cambio a 1100 pero el valor del bit de paridad fue el mismo."

- CRC32: El algoritmo es muy rápido y simple. En general siempre brinda buenos resultados en cuanto se implementa. Puede tener desventajas en tema de implementación según el grado del polinomio, pues verificar los pasos con polinomios muy altos puede ser tardado pero, en cuanto al algoritmo, tiene muchas ventajas y, de lo probado, no se encontraron desventajas.
- Hamming: Dicho algoritmo es eficiente para identificar errores de 1 switch, sin embargo, no puede corregir múltiples errores y su redundancia aumenta con el tamaño del mensaje, limitando su efectividad en entornos con altas tasas de error. En comparación a los otros algoritmos vistos en clase, no representa una mayor complejidad.