

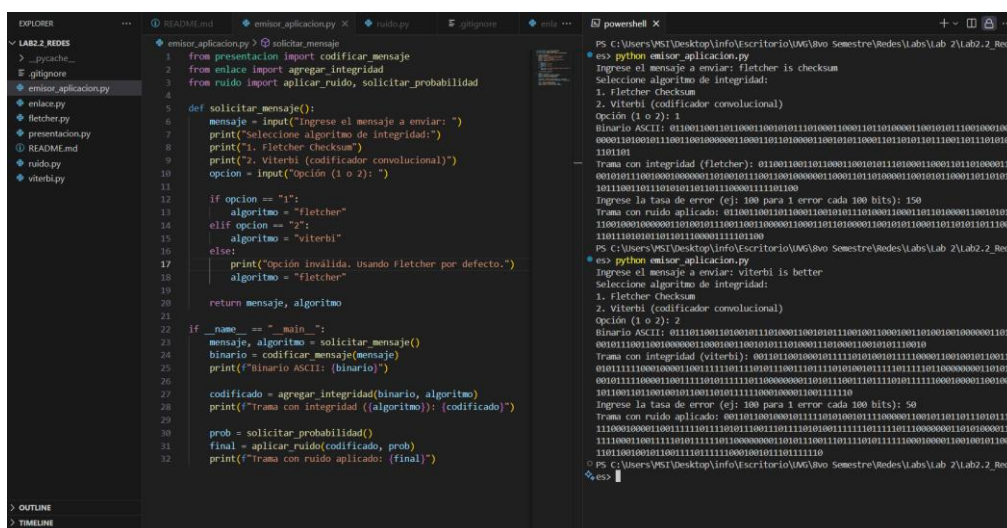
Laboratorio 2 – Parte 2: Esquemas de detección y corrección de errores

Descripción Practica:

En esta práctica se simula la transmisión de mensajes sobre un canal no confiable, utilizando una arquitectura por capas que permite aplicar esquemas de detección y corrección de errores. A partir de un mensaje ingresado por el usuario, se transforma a binario, se le aplica un algoritmo de integridad (Fletcher o Viterbi) y se simula la presencia de ruido que puede alterar los bits transmitidos. Esta entrega inicial corresponde a la implementación completa del lado del **emisor**, desde la capa de Aplicación hasta la capa de Ruido.

Funcionamiento:

El flujo del código comienza en la capa de **Aplicación**, donde el usuario ingresa el mensaje y elige el algoritmo de integridad. Luego, la capa de **Presentación** convierte el mensaje a binario ASCII. La capa de **Enlace** aplica el algoritmo seleccionado (Fletcher o Viterbi), agregando redundancia para permitir detección o corrección de errores. Finalmente, la capa de **Ruido** simula errores aleatorios en la trama, alterando bits según una probabilidad definida por el usuario. Todo el flujo se ejecuta de forma secuencial desde el script principal `emisor_aplicacion.py`.



```
emisor_aplicacion.py
1 from presentacion import codificar_mensaje
2 from enlace import agregar_integridad
3 from ruido import aplicar_ruido, solicitar_probabilidad
4
5 def solicitar_mensaje():
6     mensaje = input("Ingrese el mensaje a enviar: ")
7     print("Seleccione algoritmo de integridad:")
8     print("1. Fletcher Checksum")
9     print("2. Viterbi (codificador convolucional)")
10    opcion = input("Opción (1 o 2): ")
11
12    if opcion == "1":
13        algoritmo = "fletcher"
14    elif opcion == "2":
15        algoritmo = "viterbi"
16    else:
17        print("Opción inválida. Usando Fletcher por defecto.")
18        algoritmo = "fletcher"
19
20    return mensaje, algoritmo
21
22 if __name__ == "__main__":
23     mensaje, algoritmo = solicitar_mensaje()
24     binario = codificar_mensaje(mensaje)
25     print("Binario ASCII: {binario}")
26
27     codificado = agregar_integridad(binario, algoritmo)
28     print(f"Trama con integridad ({algoritmo}): {codificado}")
29
30     prob = solicitar_probabilidad()
31     final = aplicar_ruido(codificado, prob)
32     print(f"Trama con ruido aplicado: {final}")
```

```
PS C:\Users\PSI\Desktop\info\Escritorio\UMG\Semestre\Redes\Lab 2\Lab2.2_Red>
es> python emisor_aplicacion.py
Ingrese el mensaje a enviar: fletcher is checksum
Seleccione algoritmo de integridad:
1. Fletcher Checksum
2. Viterbi (codificador convolucional)
Opción (1 o 2): 1
Binario ASCII: 01100110011011000110010101110100011000110110100001100101011001000100
000011010010111001100100000011000110101000001100101011000110110101101101101010
1101101
Trama con integridad (fletcher): 011001100110110001100101011101000110001101101000011
0010101110010001000000110100101100110010000001100011011010000110010101100011
1011100110110101010101101110000111101100
Ingrese la tasa de error (ej: 100 para 1 error cada 100 bits): 150
Trama con ruido aplicado: 0110011001101100011001010111010001100011011010000110010101
11001000100000011010010110011001100000110001101101000110110101010110011100
110111010101010101110000111101100
PS C:\Users\PSI\Desktop\info\Escritorio\UMG\Semestre\Redes\Lab 2\Lab2.2_Red>
es> python emisor_aplicacion.py
Ingrese el mensaje a enviar: viterbi is better
Seleccione algoritmo de integridad:
1. Fletcher Checksum
2. Viterbi (codificador convolucional)
Opción (1 o 2): 2
Binario ASCII: 0111011001101001011101000110010101110010110010010010010010000001101
0010111001100100000011000100110010101101100011201000110010011110010
Trama con integridad (viterbi): 00110110010001001111010100101111000011001001010011
01011111000100001100111101110101110111011101101010101111011110110000000110101
001011110000110011110101111101100000001101011100111011101011111000100001100100
101100101110010010110011001011111000100001100111110
Ingrese la tasa de error (ej: 100 para 1 error cada 100 bits): 50
Trama con ruido aplicado: 001101100100010111110101001011100000110010110110110111
1100010000110011110111010110011001111010101111101111011000000110101000011
1111001110011110101111101100000001100111001110111010111110001000110010101100
1101100100101100111101111100010010111011110
```

