Proyecto 3 Redes : Corección y detección de errores

David Orlando De Quesada Oliva C311

Javier E. Domínguez Hernández C312

Ahora el proyecto tiene un txt configurable **config.txt** para elegir el algoritmo de detección de errores que estamos usando. Implementamos el algoritmo de **CRC** y el de **Hamming**

Si escribes en **config.txt**:

```
crc
```

aplica CRC para la detección de errores

y si escribes :

```
hamming
```

aplica el algoritmo de Hamming para la detección de errores y si puede hamming lo corrige

Agregamos también un file **signal_time.txt** para que se pueda cambiar más cómodo el **signal_time** y se tenga que cambiar a mano en el código en la variable global como estaba en las versiones anteriores.

Corregimos unos errores que tenían que ver con un error al calcular el len en bytes que se le pasaba a la trama que se nos había pasada por alto.

Para la detección de errores usando el algoritmo de Hamming hacemos lo siguiente:

CRC:

En el algoritmo de CRC está basado en la división binaria una sequencia de bits redundantes llamados called cyclic redundancy check bits (CRC) son agregados al final de la data tal que el resultado de esto sea divisible por un segundo predeterminado número binario. Cuando la información llega al destino, la unidad de datos entrantes se divide por el mismo número. Si en este paso el resto es 0 los datos llegaron de forma correcta y se acepta de lo contrario indica que la data se ha dañado durante el transporte y, por tanto, debe rechazarse y dar error.

En CRC se divide por un polinomio generador G(x) que se lleva a un polinomio binario.

Por ejemplo si

$$G(x) = x^3 + x^2 + 1 = 1x^3 + 1x^2 + 0x + 1$$

el polinomio binario seria 1101

Ahora se agrega a la data cantidad de 0 igual al len del polinomio generador en binario -1

En este ejemplo sería 000

Entonces quedaría si por ejemplo la data fuese 111101

111101**000**

Ahora se hace una división binaria de 111101**000** y 1101

Eso me da un resto ese resto es el encode del crc lo que se agrega al final de la trama

Para el decode se coge la información que debe correspondar por la posición a la data concatenada con el crc y se divide por el polinomio generador si te da resto 0 los datos llegaron de forma correcta

En el proyecto elegimos como polinomio generador a

$$x^3 + x + 1$$

que tiene representación binaria como 1011

Para la división binaria hacemos una división en galera y en cada paso en vez de restar hacemos xor entre los números como se muestra a continuación

La implementación que se hice en Python fue la siguiente:

```
def mod2div(divident, divisor):
    # Number of bits to be XORed at a time.
    pick = len(divisor)
    # Slicing the divident to appropriate
    # length for particular step
    tmp = divident[0 : pick]
    while pick < len(divident):
        if tmp[0] == '1':
            # replace the divident by the result
            # of XOR and pull 1 bit down
            tmp = xor(divisor, tmp) + divident[pick]
        else:   # If leftmost bit is '0'
            # If the leftmost bit of the dividend (or the
            # part used in each step) is 0, the step cannot
            # use the regular divisor; we need to use an
            # all-0s divisor.
            tmp = xor('0'*pick, tmp) + divident[pick]
        # increment pick to move further
        pick += 1

    if tmp[0] == '1':
        tmp = xor(divisor, tmp)
    else:
        tmp = xor('0'*pick, tmp)

    remainder = tmp
    return remainder
```

Con el método **CRCEncode**

```
def CRCEncode(data):
    l_key = len(crc_key)
    appended_data = data + '0'*(l_key-1)
    remainder = mod2div(appended_data, crc_key)

    return remainder
```

calculamos el resto de crc que se pone al final de la trama lo que le llamamos el encode

Luego cuando la trama llegue al host destino

usamos **CRCDecode**:

```
def CRCDecode(data):
    remainder = mod2div(data, crc_key)
    return remainder
```

que lo que hace es comprobar que dar el resto de la trama + encode que debe haber llegado

Si ese resto es 0 significa que todo llegó de forma correcta lo que se verifica con **CheckError**

```
def CheckError(remainder):
    return remainder == 0
```

Hamming:

El algoritmo de Hamming es uno de los algoritmos más conocidos para para detección y corrección de errores en la transmisión de datos de una fuente a un receptor. La información a transmitir en bits, es codificada usando una cierta cantidad de bits adicionales llamados bits de redundancia (o de paridad), que se usan para determinar si cuando la información llega al receptor está corrupta o no. El proceso es el siguiente:

1. Ingresar la cadena de bits a transmitir
2. Determinar la cantidad de bits de redundancia necesarios para transmitir la información
3. Determinar el valor de cada bit de redundancia

Para determinar la cantidad de bits de redundancia se usa la relación

$$2^p >= n + P + 1$$

, donde n es la cantidad de bits en la cadena a transmitir y p es el menor entero que cumple la relación antes mencionada.

Los bits de redundancia se colocan en las posiciones que son potencias de 2 en la cadena de bits a transmitir, pero hemos modificado el algoritmo para ajustarlo a los requerimientos del proyecto, donde primero viene la cadena de bits a transmitir y luego los bits de verificación, que en este caso serían los bits de redundancia. Una vez determinada la cantidad de bits de redundancia es necesario determinar el valor de dichos bits. Para ello analizaremos para cada bit de redundancia Ri, la paridad (par en este caso) de los de todos los bits en las posiciones cuya representación en binario contiene un 1 en la i-ésima posición excepto la posición de Ri, si la cantidad de 1s es par Ri = ‘0’, si es impar Ri = ‘1’. Hacemos este procedimiento con cada bit de redundancia y tenemos la cadena formada por los bits de verificación.

Detección y corrección de errores con Hamming:

Para detectar un error necesitamos comprobar que los bits de paridad (redundancia) que computamos antes de enviar la cadena de bits junto con los bits de redundancia, coincide con los bits de paridad de la cadena que recibimos en el receptor, o solo uno de los bits de redundancia es distinto, en este caso lo que estaría corrupto sería el propio bit de redundancia y la información estaría intacta. En caso de que hayan 2 o más bits de redundancia distintos, el resultado de hacer XOR entre los bits de redundancia antes de enviar la información y luego nos daría un numero en binario, ese número en decimal representa que bit de nuestros datos (de derecha a izquierda) es el que está corrupto, por tanto, lo que haríamos seria invertir su valor. Hay que aclarar que el algoritmo de hamming es capaz de corregir solo 1 error y detectar hasta 2 errores de bits consecutivos. En caso de que ningún bit de datos esté corrupto podemos obtener la cadena original, tomando solo los bits que contando de derecha a izquierda no son potencia de 2.

Los pasos que usamos al aplicar Hamming en el proyecto son:

dada la data de entrada calculamos la cantidad de bits redundantes con el siguiente método

```
def hamming_encode(data_frame):

    frame_len = len(data_frame)
    redundant_bits_amount = 0
    for i in range(frame_len):
        if(2**i >= frame_len + i + 1):
            redundant_bits_amount = i
            break

    j = 0
    k = 1
    encoded_frame = ''
    # If position is power of 2 then insert '0'
    # Else append the data
    for i in range(1, frame_len + redundant_bits_amount + 1):
        if(i == 2**j):
            encoded_frame = encoded_frame + '0'
            j += 1
        else:
            encoded_frame = encoded_frame + data_frame[-1 * k]
            k += 1

    encoded_frame = encoded_frame[::-1]
    frame_len = len(encoded_frame)

    # For finding rth parity bit, iterate over
    # 0 to r - 1
    for i in range(redundant_bits_amount):
        val = 0
        for j in range(1, frame_len + 1):
            # If position has 1 in ith significant
            # position then Bitwise OR the array value
            # to find parity bit value.
            if(j & (2**i) == (2**i)):
                val = val ^ int(encoded_frame[-1 * j])
                # -1 * j is given since array is reversed

        # String Concatenation
```

```
        # (0 to n - 2^r) + parity bit + (n - 2^r + 1 to n)
        encoded_frame = encoded_frame[:frame_len-(2**i)] + str(val) +
encoded_frame[frame_len-(2**i)+1:]

    return encoded_frame, redundant_bits_amount
```

Ese valor se lleva a binario y se agrega al final de la trama como información de verificación . La longitud de este valor en bytes se pone en los 8 bits continuos al len de la data.

Ahora luego que la información llega a su destino primero se procede a detectar si hubo error para eso se usa el siguiente método:

```
def detect_error(encoded_frame, redundant_bits_amount):
    frame_len = len(encoded_frame)
    res = 0
    # Calculate parity bits again
    for i in range(redundant_bits_amount):
        val = 0
        for j in range(1, frame_len + 1):
            if(j & (2**i) == (2**i)):
                val = val ^ int(encoded_frame[-1 * j])
        # Create a binary no by appending
        # parity bits together.
        res = res + val*(10**i)
    # Convert binary to decimal, 0 means no error
    error_index = int(str(res), 2)
    return True if error_index else False, error_index
```

En caso que haya error el va a intentar corregir usando los métodos **calc_redundant_bits** para ver el índice de error en la data sin codificar y luego con **fix_bit** lo arregla. Esta información solo se arregla como se explicaba más arriba si hay un solo error por lo que puede quedar corrompida después de haber intentado arreglarla.