Universidad Zaragoza

1542

Trabajo Fin de Grado

Análisis de tamaño, rendimiento y resistencia de algoritmos de detección y corrección de error en comunicaciones frente a fallos

Autor

Santiago Sarasa Castiella

Director

Ricardo J. Rodriguez

Escuela de Ingeniería y Arquitectura / Universidad de Zaragoza

2016

Análisis de tamaño, rendimiento y resistencia de algoritmos de detección y corrección de error en comunicaciones frente a fallos

RESUMEN

Los códigos de corrección de errores (*Error Correcting Codes*, ECC) marcaron un cambio en la transmisión de datos gracias a la confianza aportada a las comunicaciones. El emisor codifica el mensaje añadiendo bits de redundancia para que, si se produce una corrupción de los datos durante la transmisión, el receptor pueda decodificarlo, reduciendo así los errores o incluso corrigiéndolos por completo. Este sobrecoste de la transmisión asegura la integridad de los datos.

Los ECC se diferencian en dos categorías según si realizan o no la retransmisión del mensaje en caso de error. Este Trabajo de Fin de Grado se ha centrado en las técnicas que no necesitan retransmisión, llamados *Forward Error Correction* (FEC).

Tomando en consideración estos aspectos, en este trabajo se han implementado diferentes técnicas FEC que codifican y reducen los errores producidos (en concreto, Reed-Solomon, Convolucional, Turbo Código y Low Density Parity Check). Se han realizado experimentos para comprobar el rendimiento de estas técnicas frente a la inserción de error, suponiendo un escenario donde se produce una transmisión de datos durante la cual se añade ruido, produciendo errores de manera aleatoria.

Los resultados muestran que las técnicas implementadas funcionan correctamente y son aplicables en un entorno de error de bit aleatorio. Las técnicas que consiguen mejores resultados son las más actuales, Turbo Código y *Low Density Parity Check*, llegando hasta prácticamente la total corrección de un error de bit aleatorio inferior al 13 %, a cambio de un alto coste computacional (con un sobrecoste del 2231 % para el fichero de menor tamaño). Los códigos desarrollados se han liberado para su utilización por terceros libremente y para garantizar la reproducibilidad de los resultados obtenidos.

Índice

Índice de Figuras v

Índice de Tablas vii

1. Introducción 1
- 1.1. Contexto . . . 1
- 1.2. Motivación . . . 1
- 1.3. Objetivos del proyecto . . . 2
- 1.4. Estructura del documento . . . 2

2. Conocimientos previos 3
- 2.1. Códigos de Corrección de Errores (ECC) . . . 3
- 2.2. Reed-Solomon . . . 5
- 2.3. Low Density Parity Check . . . 7
- 2.4. Convolucional . . . 9
- 2.5. Turbo Código . . . 11
- 2.6. Lenguaje Unificado de Modelado . . . 13

3. Trabajo relacionado 17

4. Análisis y Diseño 19
- 4.1. Arquitectura Software . . . 19
- 4.2. Reed-Solomon . . . 21
- 4.3. Low Density Parity Check . . . 24
- 4.4. Convolucional . . . 28
- 4.5. Turbo Código . . . 30
- 4.6. Disponibilidad . . . 34

5. Evaluación y resultados 35

6. Conclusiones 41

Bibliografía 43

Acrónimos **45**

A. Horas de trabajo **46**

Índice de Figuras

2.1. Redundancia de bits. 3
2.2. Comparación de rendimiento de las técnicas (extraída de [CCS12]). . . . 5
2.3. Proceso de codificación RS. 6
2.4. Proceso de decodificación RS. 6
2.5. Gráfico de Tanner para una matriz H. 8
2.6. Proceso de decodificación LDPC. 8
2.7. Generador de secuencias de $K = 3$, $k = 1$ y $n = 2$ con $G_1 = 7$ y $G_2 = 5$. . 9
2.8. Diagrama de estados para $K = 2$, $k = 1$ y $n = 2$ (extraído de [MSM14]). . 10
2.9. Diagrama de Trellis para $K = 2$, $k = 1$ y $n = 2$ (extraído de [MSM14]). . 10
2.10. Sistema Turbo Código con 2 codificadores, $r = \frac{1}{3}$. 11
2.11. Punteado de $r = \frac{1}{3}$ a $r = \frac{1}{2}$. 11
2.12. Estructura decodificador TC (extraído de [VA10]). 12
2.13. Diagrama de clases. 14
2.14. Diagrama de secuencia. 14

4.1. Diagrama de clases de la implementación. 20
4.2. Diagrama de secuencia de Reed-Solomon. 22
4.3. Diagrama de secuencia del LDPC. 25
4.4. Diagrama de secuencia del código Convolucional. 30
4.5. Diagrama de secuencia del Turbo Código. 33

5.1. Escenario experimentación. 35
5.2. Comparativa tiempos de codificación por código. 37
5.3. Comparativa tiempos de decodificación por código. 37
5.4. Comparativa tiempos de decodificación según número iteraciones. 38
5.5. Comparativa BER vs Error introducido por código. 39
5.6. Comparativa BER vs Error introducido por iteración. 39

A.1. Diagrama de Gantt. 46

Índice de Tablas

5.1. Tabla comparación de tamaño codificado. 36

A.1. Duración de cada tarea. 47

Capítulo 1

Introducción

En este primer capítulo se presentan los conceptos fundamentales que se trabajan a lo largo de la memoria. En primer lugar, se ponen en contexto la teoría de codificación así como los Códigos de Corrección de Errores (*Error Correcting Code*, ECC). Una vez mostrada el área de aplicación, se presentan las motivaciones y los objetivos a alcanzar. Finalmente, se describe brevemente la organización y contenido de la memoria para ayudar al lector a ubicar con claridad los capítulos.

1.1. Contexto

Este proyecto está relacionado con la teoría de codificación [Sha48] y la corrección de errores producidos durante la transmisión. La teoría de codificación se basa en el estudio del canal de transmisión y conseguir que, a través del canal, se produzca una comunicación de confianza (entendiéndose por confianza la capacidad por la cual los mensajes alcanzan su destino e íntegros).

En segundo lugar, se debe tener en cuenta el constante aumento de dispositivos con la necesidad de transmitir cada vez más datos, más rápido y con menos errores. Para conseguir un alto grado de fiabilidad durante la transmisión, es necesario conocer los ECC y su rendimiento en función de las características de la comunicación y los errores del canal.

1.2. Motivación

Este TFG se centra en los ECC, en concreto, en las técnicas de Corrección de Errores hacia Adelante (*Forward Error Correction*, FEC) para detección de errores por varios motivos. En primer lugar, porque se quiere enfocar este trabajo a las comunicaciones sin garantía de entrega y con dificultades de retransmisión. En segundo lugar, porque se distribuye la tarea de corrección entre el emisor y receptor, haciendo que una vez recibido el mensaje desaparezca la dependencia del canal.

Estas capacidades presentan un contexto idóneo para pensar en las necesidades de comunicación en espacio profundo, obteniendo un mecanismo de confianza sobre la fia-

bilidad de las transmisiones. De hecho, las técnicas FEC son una de las soluciones de mayor relevancia a día de hoy, siendo ya utilizadas por la NASA en las comunicaciones durante sus misiones [CCS12].

1.3. Objetivos del proyecto

El objetivo de este proyecto es estudiar, implementar y comparar diferentes técnicas FEC frente a errores aleatorios. El proceso que se ha seguido es el siguiente:

- Estudiar qué es un ECC, sus tipos y cómo funciona.
- Estudiar los diferentes tipos de códigos FEC.
- Diseñar con UML e implementar en Java las técnicas FEC elegidas.
- Experimentar las capacidades de las técnicas en un entorno controlado.
- Evaluar el alcance de la solución y su viabilidad.

Este proceso ha dado como resultado un sistema de codificación de comunicaciones para evitar la corrupción de los datos en la transmisión. El sistema dispone de cuatro códigos FEC para la codificación y posterior decodificación del mensaje enviado. Con este prototipo se ha analizado el rendimiento de las técnicas en un escenario de adición de ruido durante la transmisión. Los resultados obtenidos han validado la implementación, demostrando que funciona correctamente.

1.4. Estructura del documento

El presente documento está dividido en dos partes: la memoria, donde se explica el desarrollo del TFG; y el apéndice, donde se amplía la información respecto al trabajo realizado.

El Capítulo 2 define los conocimientos previos necesarios para comprender este trabajo en su totalidad. El Capítulo 3 muestra los trabajos relacionados con este TFG. El Capítulo 4 explica las técnicas elegidas, tanto a nivel conceptual como a nivel de su implementación. El Capítulo 5 muestra y analiza los resultados obtenidos. Finalmente, en el Capítulo 6 se exponen las conclusiones obtenidas. Respecto al anexo, el Apéndice A muestra el balance del esfuerzo invertido en este TFG.

El código se ha liberado bajo la licencia GNU GPLv3 y puede encontrarse en el repositorio de:

`https://bitbucket.org/ssarasa/fectechniques`

Capítulo 2

Conocimientos previos

En este capítulo se describe con mayor detalle el entorno sobre el que se desarrolla el TFG, explicando qué son los Códigos de Corrección de Errores y sus tipos; qué es la Correción de Errores hacia Adelante (del inglés, *Forward Error Correction*, FEC), y las diferentes técnicas implementadas; y el Lenguaje Unificado de Modelado (*Unified Modeling Language*, UML).

2.1. Códigos de Corrección de Errores (ECC)

Los códigos de corrección de errores son un concepto utilizado en la trasmisión de datos digitales, siendo objeto de investigación y nuevas implementaciones desde que nació la teoría de codificación.

Estas técnicas se basan en el proceso denominado *Codificación de Canal*, por el que se añaden bits adicionales al mensaje de *información* como *redundancia* por el emisor para identificar y corregir posibles errores ocurridos en la transmisión, como se puede ver en la Figura 2.1. Así se consigue el envío de mensajes válidos a través de canales no confiables, donde se pueden producir errores durante la transmisión en forma de cambios en los datos enviados. El número de bit erróneos respecto al numero de bits transmitidos se llama tasa de error de bit (del inglés *Bit Error Rate*, BER).

Estos métodos se dividen en dos categorías:

- **Automatic Repeat Request (ARQ)**: Se basa principalmente en la detección de errores y retransmisión. Cuando se detecta un error en la recepción, se retrans-

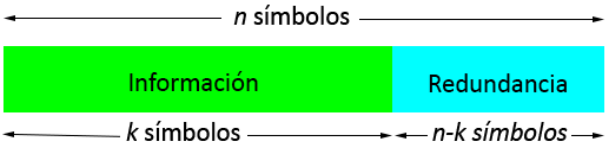

Figura 2.1: Redundancia de bits.

mite la trama errónea hasta que ésta se recibe correctamente. Debido al retardo introducido por la retransmisión, se utiliza en aplicaciones no basadas en tiempo real.

- **Forward Error Correction (FEC)**: Se basa principalmente en la corrección de errores, aunque algunas de las técnicas también proporcionan detección de errores. Los bits añadidos son bits de paridad que permiten recuperar la mayor cantidad de información posible a pesar de que haya habido errores en la transmisión. Las técnicas implementadas en este trabajo pertenecen a esta categoría.

  Estos códigos se suelen separar en dos grupos:

  - **Códigos de bloque**: Estos códigos no tienen memoria, siendo el resultado de la operación de codificación aplicada producido en un único ciclo. Para la entrada de un bloque de k símbolos, se produce un bloque de salida de n símbolos. Por ello, se representan mediante los parámetros (n,k).

    Los códigos implementados en este TFG de este grupo son Reed-Solomon (RS) [Uni10b, Uni10a] y Low Density Parity Check (LDPC) [MG10], explicados con mayor detalle en las Secciones 2.2 y 2.3, respectivamente.
  - **Códigos convolucionales**: A diferencia de los anteriores, estos códigos tienen memoria, ya que cada bit de la trama codificada depende no sólo del bit actual, sino también de los bits anteriormente introducidos en el codificador. Los parámetros que representan este tipo de códigos son K, la cantidad de estados de memoria en el codificador, y r, la tasa de codificación (indica el números de bits de salida por cada bit de entrada), especificándose entonces como (K,r).

    Los códigos implementados en este TFG de este grupo son Convolucional [MSM14] y Turbo Code (TC) [Skl97], explicados con mayor detalle en la Sección 2.4 y la Sección 2.5, respectivamente.

  Gracias a la aplicación de estos códigos se optimizan las capacidades de las antenas de la comunicación, ya que al reducir el BER y la tasa de error es necesaria menor potencia de transmisión – siempre sin superar el Límite de Shannon [Kol13], que representa la tasa máxima para el envío de datos con error cero. Este límite varía según el ancho de banda del canal y las características del ruido.

  Para la elección de los parámetros de cada uno de los códigos FEC seleccionados se han consultado los estándares del CCSDS (*Consultative Committee for Space Data Systems*) [CCS11, CCS12]. En la Figura 2.2 (extraída de [CCS12]) se observan los códigos seleccionados frente al Límite de Shannon. Por limitaciones computacionales en el TFG el código LDPC está formado por bloques 1024 bits, en lugar de 16384 bits.

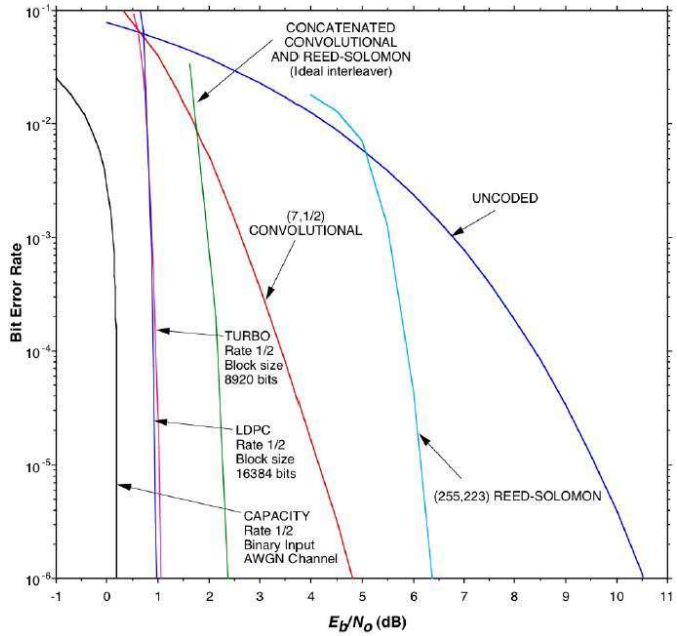

Figura 2.2: Comparación de rendimiento de las técnicas (extraída de [CCS12]).

2.2. Reed-Solomon

El código Reed-Solomon (RS) es un FEC que utiliza bloques no binarios para corregir errores aleatorios y errores de ráfaga. En los errores aleatorios el error se produce de forma individial (un bit), mientras que en los errores de ráfaga el error se produce en todo un bloque de bits.

Al utilizar bloques no binarios la unidad de información que maneja es el símbolo, cuya longitud depende del tamaño del bloque codificado de la forma $n = 2^m - 1$, siendo m la longitud del símbolo en bits.

Para cada bloque de k símbolos de entrada al codificador se producen n símbolos de salida, véase la Figura 2.3, añadiendo $2t = n - k$ símbolos de redundancia. Esto permite recuperar hasta t símbolos erróneos. Para ello, es necesario utilizar el Campo de Galois.

El Campo de Galois o *Galois Field* (GF) [ISR60] es un cuerpo finito en el cual se producen las operaciones matemáticas utilizadas en el Reed-Solomon. Los símbolos utilizados se definen sobre el campo GF(2^m). A partir del generador del campo se construye todo el alfabeto, que está compuesto por 2^m palabras. Cada una de estas palabras se expresa de la forma α^i, $i \in \{0 \dots 2^m - 1\}$, y es un polinomio binario que corresponde a un símbolo del Reed-Solomon. En este campo existen dos operaciones: la suma, que es una suma binaria de los coeficientes del polinomio; y la multiplicación, que consiste en la suma modular de los exponentes de la palabra.

Codificación

Se basa en la utilización de un diccionario pequeño de palabras pero muy bien seleccionadas, donde cada palabra está separada por una distancia mínima de $2t + 1$. Esta distancia es el número de letras diferentes que tienen dos palabras del diccionario. El polinomio generador para este diccionario es el producto de los factores $(x - \alpha^n)$, siendo $n \in \{1 \dots m + 1\}$.

El bloque introducido en el codificador es operado con el polinomio generador del diccionario y se le añade el resto de esta operación, que es único ya que el polinomio generador es irreducible. Este resto será la parte de redundancia del código que se utilizará para la recuperación del mensaje original.

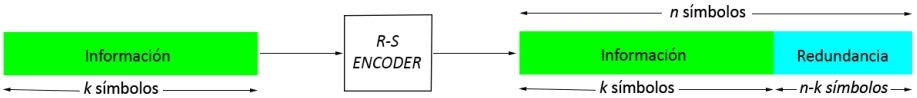

Figura 2.3: Proceso de codificación RS.

Decodificación

Utiliza el bloque previamente generado en la codificación para la detección y reparación de errores. Este proceso se basa en 4 pasos, como se puede ver en Figura 2.4:

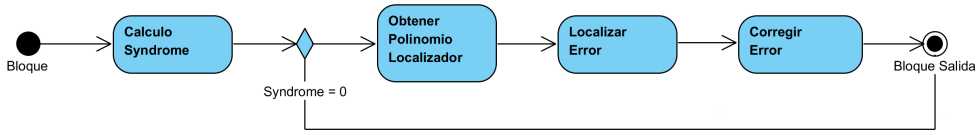

Figura 2.4: Proceso de decodificación RS.

1. **Calcular syndrome** S: El syndrome S del bloque es un polinomio de control que indica si el mensaje es correcto. Para calcularlo se trata el mensaje en forma de polinomio y se evalúa en las raíces del polinomio generador $(\alpha^0, \alpha^1, \alpha^2, ..., \alpha^n)$. Si S está formado por todo ceros es que el mensaje es correcto; si no, es que el bloque contiene algún error.

2. **Obtener polinomio localizador** Z: A partir de S se obtiene el polinomio localizador de error Z, que indica qué valores hay corruptos. Esto se realiza mediante el Algoritmo de Berlekamp-Massey [Can11]. Este algoritmo busca el *Lineal Feedback Shift Register* (LFSR, registro de desplazamiento con retroalimentación lineal) más corto que reproduce el syndrome del bloque.

3. **Localizar el error**: Se comprueba sobre Z, mediante fuerza bruta, todos los posibles valores. Los valores que sean cero indican las posiciones del error.

4. **Corregir el error**: Teniendo S y la localización del error, se aplica el Algoritmo de Forney [For65] para corregir hasta t símbolos erróneos.

   Este algoritmo realiza el cálculo de los valores de σ_i $i \in \{1 \dots t\}$ con los coeficientes de Z:

$$\sigma_i = \prod_{j=0|j\neq i}^{t-1} (x + Z_j) = \sigma_{i,0} \cdot x^{t-1} + \sigma_{i,1} \cdot x^{t-2} + ... + \sigma_{i,t-1} \tag{2.1}$$

   donde $\sigma_{i,j}$ es el coeficiente para obtener la magnitud del error i y x^k es la variable del polinomio. Una vez obtenido $\sigma_{i,j}$, y utilizando también los valores de S, se obtiene la corrección del error mediante la fórmula:

$$Y_i = \frac{\sum_{j=0}^{t-1}(S_j \cdot \sigma_{i,t-1-j})}{Z_i \cdot \prod_{j=0|j\neq i}^{t-1}(Z_i + Z_j)} \tag{2.2}$$

   Finalmente, se aplican las correcciones obtenidas a las posiciones erróneas del bloque.

2.3. Low Density Parity Check

El código *Low Density Parity Check* (LDPC) es un código FEC de bloque que basa su funcionamiento en el uso de una matriz de paridad dispersa (baja densidad de unos). Utiliza un proceso de decodificación iterativo de intercambio de información entre los nodos que componen la matriz, para una correcta estimación del bit de información.

Se caracteriza mediante los parámetros (n,k), donde k suele ser un bloque de tamaño grande. En este trabajo el bloque está limitado por capacidades computacionales. El rendimiento del código está cercano al Límite de Shannon [MN96].

Codificación

Utiliza una matriz de paridad H_r, compuesta por submatrices de tamaño MxM, con $M = k/2$ para $r = 1/2$. Para generar esta matriz se deben seguir las indicaciones propuestas por el estándar CCSDS [CCS11].

Mediante operaciones matriciales a partir de la matriz H_r se puede obtener una matriz de alta densidad (llamada matriz de generación G_r), que será la matriz con la que se codificará el bloque. Esta matriz está compuesta por una matriz identidad $2M$x$2M$ y en el resto, la matriz de redundancia correspondiente a H_r.

Para el proceso de codificación se multiplica el bloque por G_r, por lo tanto la primera parte del bloque codificado es el bloque de información. Las últimas M columnas del bloque codificado son punteadas: en lugar de enviar las $5M$ columnas producidas, el bloque codificado se reduce a $4M$ para ajustarlo a la tasa del código.

Decodificación

Se basa en decodificación iterativa en la que los Nodos de Comprobación (*Check Nodes*, CN) intercambian información con los Nodos Variables (*Variable Nodes*, VN), que estiman el valor correcto para cada bit. Los CN corresponden a las filas de la matriz H_r y se encargan de corregir el valor del bit con la información aportada por los VN; mientras que los VN corresponden a las columnas de la matriz H_r y se encargan de comprobar el valor recibido con el resto de *Check Nodes* y reajustarlo si es preciso. En la Figura 2.5 se puede ver un Gráfico de Tanner, con el cual se puede expresar gráficamente la matriz y las relaciones entre los nodos.

$$H = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

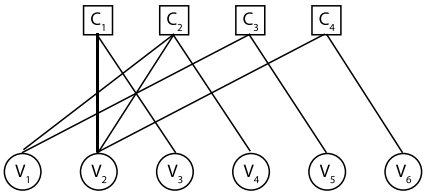

Figura 2.5: Gráfico de Tanner para una matriz H.

El intercambio de mensajes entre los nodos y el cálculo de probabilidades es un proceso iterativo que finaliza cuando el mensaje es correcto o cuando se alcanza el número máximo de iteraciones especificadas en el código. Este proceso se muestra en 2.6, y consta de los siguientes pasos:

1. **Inicialización**: Calcula la probabilidad del valor del bit en función del bloque recibido. Este valor es con el que se inicializan los CN.

2. **Cálculo del syndrome**: Obtiene el syndrome del bloque operándolo con la matriz H. Si el syndrome está formado por ceros es que el bloque es correcto y se detiene el proceso de iteración, en caso contrario continúa.

3. **Cálculo en VN**: Los CN envían los valores calculados a los VN. Con la información aportada por todos los CN a los que están conectados, los VN calculan la probabilidad de que el valor recibido de cada CN cumpla la paridad del nodo.

4. **Cálculo en CN**: Los VN envían los valores estimados a los CN, que calculan la probabilidad del valor del bit en función del bloque recibido y los valores de los VN.

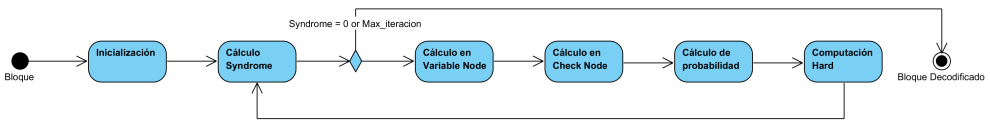

Figura 2.6: Proceso de decodificación LDPC.

5. **Cálculo de probabilidad**: Calcula para cada bit, con los datos aportados por los nodos, la probabilidad de que el bit tenga valor 1.

6. **Computación hard**: Realiza para todos los bits una decisión *hard*, de modo que si la probabilidad es mayor que 0.5 asigna al bit el valor 1; en caso contrario, asigna al bit el valor 0.

2.4. Convolucional

El código Convolucional es un FEC que trabaja directamente con el flujo completo de información y que tiene memoria. Estos códigos son menos sensibles a la variación de relación señal-ruido que los códigos de bloque y se suelen utilizar en aplicaciones en las que es necesario obtener un buen rendimiento con un coste de implementación bajo.

Los K estados de la memoria del codificador están estructurados mediante un LFSR; los valores k y n del código normalmente suelen ser valores bajos para que haya poco retardo en decodificación.

Codificación

Se basa en la utilización de un generador de secuencias como el que se muestra en la Figura 2.7, donde se van introduciendo k bits cada vez desplazando los K estados de memoria del codificador y se calculan los n bits de salida, cada uno producido por un polinomio generador. Al generarse de esta manera el flujo de salida, los bits presentes no guardan relación con el flujo de entrada, pero sí que sirven como redundancia de éste. Esta operación se realiza hasta que se ha introducido el flujo completo de información.

Cada vez que se produce una entrada de k bits, el sistema cambia de estado generando una salida de n bits. Para una mayor comprensión de las operaciones internas realizadas por el código se suelen utilizar diagramas de estado, que se representan como una máquina de estados de Mealy [Mea55], véase la Figura 2.8. Este tipo de diagramas muestra las transiciones entre estados, cuándo se activan y qué bits de salida producen. De esta manera, se puede seguir el comportamiento del generador de secuencias y prever el flujo de salida que se codificará.

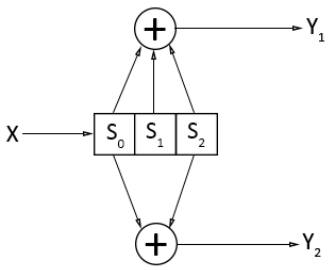

Figura 2.7: Generador de secuencias de $K = 3$, $k = 1$ y $n = 2$ con $G_1 = 7$ y $G_2 = 5$.

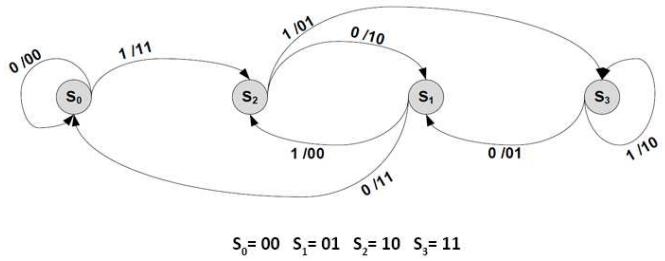

Figura 2.8: Diagrama de estados para $K = 2$, $k = 1$ y $n = 2$ (extraído de [MSM14]).

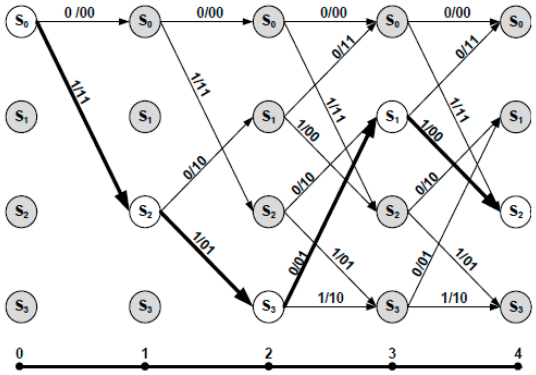

Figura 2.9: Diagrama de Trellis para $K = 2$, $k = 1$ y $n = 2$ (extraído de [MSM14]).

Decodificación

Para la decodificación de códigos convolucionales se utiliza el Algoritmo de Viterbi [Vit67]. Este algoritmo compara el flujo recibido con todas las posibles codificaciones hasta encontrar el de máxima similitud. Para ello, es necesario realizar el Diagrama de Trellis [BCJR74] del código. En este diagrama se muestra de una manera gráfica frente a una línea temporal lo expuesto en el diagrama de estados, donde la unidad de tiempo es una transición de estado. En la Figura 2.9 se puede ver un Diagrama de Trellis para $K = 2$, $k = 1$ y $n = 2$.

Para la aplicación del algoritmo se compara el flujo recibido en divisiones de n bits frente a la rama correspondiente en el Diagrama de Trellis, midiendo la similitud entre ambos. En el caso de que dos ramas vayan a dar al mismo nodo, el algoritmo se queda con la rama que tenga la métrica acumulada más grande, puesto que la similitud de todo el camino frente al flujo es mayor. Finalmente, el camino que tenga mayor métrica será el flujo corregido – o al menos, el más próximo a una mejor corrección.

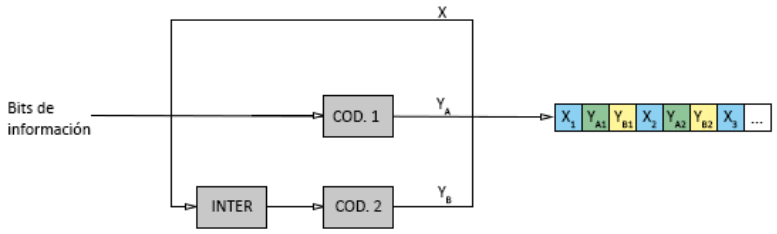

Figura 2.10: Sistema Turbo Código con 2 codificadores, $r = \frac{1}{3}$.

2.5. Turbo Código

El Turbo Código (*Turbo Code*, TC) es un FEC que, aunque utiliza bloques para la división del mensaje, realiza la codificación mediante códigos convolucionales. Por ello, pertenece a la categoría de los FEC convolucional. Este código utiliza un algoritmo de decodificación iterativo para realizar la corrección del bloque.

La caracterización de este código es (r, k), siendo r la tasa de codificación del bloque y k la longitud del bloque en bits; aunque para poder conocer por completo el sistema también es necesario aportar los datos de los codificadores convolucionales.

Codificación

Se basa en el uso de diferentes codificadores convolucionales que reciben como entrada el mismo bloque, pero para generar diferentes codificaciones, los bits del bloque son reordenados de acuerdo al entrelazador existente delante de cada codificador. En la Figura 2.10 se muestra un ejemplo de un TC con dos codificadores y $r = 1/3$.

Para formar el bloque codificado es necesario llevar $N+1$ ramas en paralelo, donde la primera rama es el bloque de entrada y las N siguientes son el número de codificadores que tiene el sistema. De estas ramas, se toman los bits de manera ordenada para formar el bloque, quedando en la primera posición el primer bit del bloque codificado y en los N siguientes bits el primer bit de cada uno de los codificadores, repitiéndose este orden hasta el final del bloque.

Según la tasa r, es necesario realizar un punteado en las salidas de las ramas: en lugar de tomar el bit de los N codificadores, se toma alternativamente el bit de cada codificador para ajustarlo a r. En la Figura 2.11 se muestra un ejemplo de un punteado de $r = 1/3$ a $r = 1/2$.

Figura 2.11: Punteado de $r = \frac{1}{3}$ a $r = \frac{1}{2}$.

Decodificación

A diferencia de los códigos anteriores que utilizan un tipo de decodificación *Hard*, en el Turbo Código se utiliza una decodificación *Soft*. Este tipo de decodificación utiliza un rango mayor de posibles valores para indicar la fiabilidad del dato, realizando mejores estimaciones sobre la información original.

Se basa en decodificadores individuales para cada uno de los N codificadores, véase la Figura 2.12. Cada uno de estos decodificadores recibe 3 flujos distintos: la parte del bloque de la información original x'_k, la parte de paridad del bloque correspondiente al codificador p'_{ik}, y la estimación realizada por el decodificador anterior $Le^i(x_k)$. Estos flujos los usa para realizar una nueva estimación entre los bits de información y los codificados. El decodificador del TC realiza tantas iteraciones como especifique el usuario.

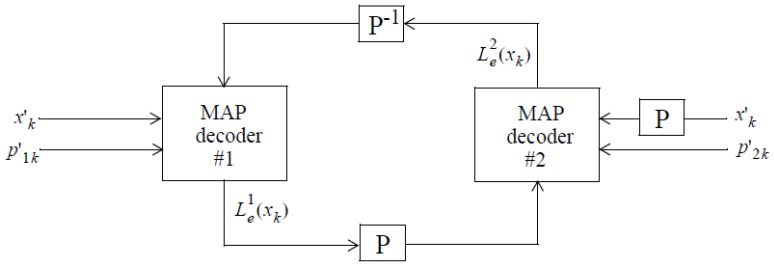

Figura 2.12: Estructura decodificador TC (extraído de [VA10]).

Los decodificadores convolucionales utilizan un algoritmo de *Maximum A-posteriori Probability* (MAP) [BDMP96], que emplea todos los bits del flujo para obtener el bit más probable en cada paso del Diagrama de Trellis. Esta decisión MAP se define como:

$$L_{map}(x_k) = L_a(x_k) + L_c \cdot x_k + L_e(x_k) \tag{2.3}$$

donde L_a es la estimación del bit k del decodificador anterior; $L_c = 2/\sigma$, con σ como la desviación estándar del ruido; x_k es el bit de información k; y $L_e(x_k)$ es la estimación del decodificador. El cálculo $L_e(x_k)$ se define como:

$$L_e(x_k) = log[\frac{P(x_k = +1)}{P(x_k = -1)}] \tag{2.4}$$

siendo estos valores la probabilidad de que el bit en la posición k tenga el valor *soft* $+1$ (correspondiente a bit en 1) y -1 (correspondiente a bit en 0). Mediante el teorema de Bayes se obtiene:

$$\frac{P(x_k = +1)}{P(x_k = -1)} = \frac{\sum_{(s',s)\in S^+} (\alpha_{k-1}(s') \cdot \gamma_k(s',s) \cdot \beta_k(s))}{\sum_{(s',s)\in S^-} (\alpha_{k-1}(s') \cdot \gamma_k(s',s) \cdot \beta_k(s))} \tag{2.5}$$

donde $\gamma_k(s',s) = \exp[\frac{1}{2}(x_k L_a(x_k) + x_k L_c x'_k + p_k L_c p'_k)]$ indica la probabilidad de transición del estado s' a s en la posición k, calculado mediante la información recibida por el

decodificador; $\alpha_k(s) = \sum_{(s')}(\alpha_{k-1}(s') \cdot \gamma_k(s', s))$ indica la probabilidad de que el bit en la posición k se encuentre en el estado s y se obtiene mediante la suma de todas las ramas que llevan al estado; y $\beta_k(s') = \sum_{(s')}(\gamma_{k+1}(s', s) \cdot \beta_{k+1}(s))$ indica la probabilidad de que el bit en la posición k salga hacia el estado s, y se obtiene mediante la suma de todas las ramas que finalizan el Trellis desde ese estado.

El decodificador devuelve la estimación $L_{map}(x_k)$ obtenida como información extrínseca para el siguiente decodificador y el cálculo final de la corrección.

2.6. Lenguaje Unificado de Modelado

El Lenguaje Unificado de Modelado o UML [RJB04], es un lenguaje gráfico para describir el sistema mediante modelos. Está estandarizado por ISO y respaldado por el *Object Management Group* (OMG).

Es un lenguaje que permite abstraerse del lenguaje de implementación para enfocarse en los esquemas de diseño, proporcionando apoyo durante las fases de desarrollo del sistema.

UML cuenta con varios tipos de diagramas que representan un aspecto diferente del modelado y que se complementan para la caracterización completa del sistema. Se pueden organizar en las siguientes categorías:

1. **Diagramas de estructura**: Se basan en el modelado de que debe estar presente en el sistema. Se utilizan mucho en la arquitectura de software. Esta categoría incluye los diagramas de clases, componentes, objetos y despliegue.

2. **Diagramas de comportamiento**: Se basan en el modelado de lo que debe suceder en el sistema, y describen la funcionalidad del sistema. Esta categoría incluye los diagramas de actividades, casos de uso y máquina de estados.

3. **Diagramas de interacción**: Son un subgrupo de los diagramas de comportamiento; se basan en el flujo de control y datos dentro del sistema. Esta categoría incluye los diagramas de secuencia, comunicación, vista de interacción y tiempo.

En este TFG se han utilizado el Diagrama de clases, el Diagrama de secuencia y el Diagrama de flujo.

Diagrama de clases

El diagrama de clases muestra las clases existentes en el sistema, sus atributos y métodos, además de las relaciones y herencias existentes entre cada una. En la Figura 2.13 se puede ver un ejemplo de este diagrama.

Para el diseño de este diagrama se implementa cada clase mediante un rectángulo dividido en tres partes: en la primera se especifica el nombre de la clase, en la segunda sus atributos y en la última sus métodos. Delante de cada atributo y método se especifica

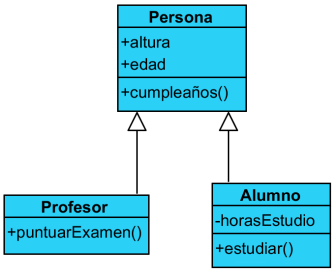

Figura 2.13: Diagrama de clases.

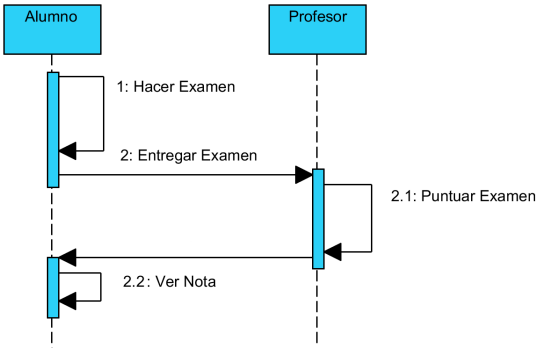

Figura 2.14: Diagrama de secuencia.

si es público (+) o privado (-). La herencia entre clases se especifica con una flecha vacía que apunta a la clase madre de la relación.

Diagrama de secuencia

El diagrama de secuencia muestra los objetos y las clases del sistema y los intercambios de mensajes que se producen entre ellos a través del tiempo, de esta manera se puede seguir el diagrama y ver qué objetos intervienen durante un momento concreto de la ejecución y el comportamiento que deberían tener. En la Figura 2.14 se puede ver un ejemplo de este diagrama.

Para el diseño de este diagrama se implementan verticalmente de forma paralela las líneas de vida de los objetos y de manera horizontal el intercambio de mensajes, apareciendo ordenados cronológicamente. Estos mensajes pueden ser síncronos, si espera hasta que recibe la respuesta del mensaje; o asíncronos, si continúa sin esperar respuesta. El tiempo que un objeto está ocupado con una tarea se representa con un rectángulo.

Diagrama de actividad

El diagrama de actividad muestra las actividades que se producen durante la ejecución de un algoritmo en el sistema. En la Figura 2.4 de la Sección 2.2 se puede ver un ejemplo de este diagrama.

Para el diseño de este diagrama se implementan las actividades mediante rectángulos que se unen según su orden de ejecución en el algoritmo. Las operaciones de decisión se representan a través de un rombo en el cual está descrita la salida afirmativa. Por último, mediante un círculo completo y dos círculos concéntricos se representan el inicio y final del algoritmo, respectivamente.

Capítulo 3

Trabajo relacionado

Para el desarrollo de este TFG ha sido necesaria la lectura y comprensión de diferentes trabajos relacionados, principalmente sobre la base teórica e implementación de las técnicas FEC seleccionadas.

En [CCS12] se presenta el estándar del CCSDS para la *codificación de canal y sincronización de telemetría*. Explica las técnicas FEC utilizadas en sistemas espaciales y las posibles configuraciones de sus parámetros, mostrando ventajas y desventajas de utilizar cada una de ellas; aparte, realiza comparaciones de rendimiento frente a la relación señal-ruido del canal. En [CCS11] se puede ver los parámetros técnicos recomendados por el estándar para la implementación de las técnicas FEC.

En [MSM14] se presenta la utilización de los códigos convolucionales para las redes de RTA (Aplicaciones en Tiempo Real), explicando qué son y los pasos para su implementación. Entre las diferentes técnicas de decodificación expuestas, se encuentra el algoritmo de Viterbi y se puede comprobar su comparación en complejidad de computación frente a la decodificación secuencial.

En [ZZL13] se presenta los principios de decodificación del Turbo Código y varias técnicas para realizar esa decodificación. Finalmente, se muestra diferentes simulaciones variando los parámetros del TC en un canal AWGN, para obtener el rendimiento del código en el canal.

En [RSU01] se realiza una visión general del código LDPC y se muestra una comparación de diferentes familias del código en función de su capacidad correctora, mostrando además su complejidad computacional y arquitectura.

Todas las comparaciones de rendimiento consultadas en estos trabajos presentan los resultados respecto al canal en el que se produce la comunicación, obteniendo resultados en función de las características del canal. En este trabajo se enfocan estas comparaciones al rendimiento de las implementaciones de las técnicas a nivel software, obteniendo valores de las capacidades de las técnicas en función del error presente.

Capítulo 4

Análisis y Diseño

En primer lugar, en este capítulo se detalla cuál es la arquitectura software de la estructura implementada. A continuación, se analiza conceptualmente cada una de las técnicas y se explica su funcionamiento a nivel de implementación.

4.1. Arquitectura Software

El conjunto de los códigos de corrección de errores propuestos se han implementado en Java con el apoyo de Matlab para el cálculo de las matrices H_r y G_r del LDPC, y comprobación de resultados en operaciones matriciales.

Para describir el funcionamiento de la implementación en primer lugar se debe hacer referencia al Diagrama de Clases presente en la Figura 4.1 que muestra cómo se relacionan las diferentes clases presentes en el sistema y sus métodos públicos o abstractos (se han omitido los métodos privados por claridad). El diseño se ha realizado siguiendo el patrón de diseño estructural *Facade* para simplificar el sistema y reducir su complejidad, proporcionando una clase unificada para el conjunto de clases del sistema

La clase **ForwardErrorCorrection** es la clase abstracta de la que heredan el resto de códigos. Contiene los métodos abstractos *encode()* y *decode()*, que serán implementados por sus clases hijas. Contiene también las funciones de generación de error usadas en la experimentación y que se explican en la Sección 4.1. **Galois** es la clase que genera un campo de Galois, usado por el Reed-Solomon. **TurboDecoder** es la clase que implementa cada decodificador individual de la decodificación iterativa del Turbo Código. El resto de clases son las clases propias para la ejecución de cada una de las técnicas.

Para explicar en mayor detalle el funcionamiento del sistema se analizan por separado las técnicas implementadas. En cada una de las técnicas se presenta un diagrama de secuencia, que muestra la interacción del código, y el algoritmo de funcionamiento en pseudocódigo, detallando el funcionamiento de las partes más importantes.

Figura 4.1: Diagrama de clases de la implementación.

4.2. Reed-Solomon

El diagrama de secuencia de la Figura 4.2 muestra las funciones públicas a las que se puede invocar: la instanciación y las funciones *encode()* y *decode()*. Durante la instancia se reciben los parámetros n, k, los bits por símbolo y el polinomio generador, y se genera un objeto de la clase **Galois** para los cálculos en el sistema. En las funciones *encode()* y *decode()* se divide el mensaje en bloques en función de k y n, respectivamente, para que las funciones *encodeBlock()* y *decodeBlock()* trabajen sobre cada bloque.

En el Algoritmo 1 se explica cómo funciona el *decodeBlock()* en mayor detalle. Para cada bloque que recibe la función se calcula el syndrome S (líneas 1 a 3). Si S tiene algún valor no cero (líneas 4 a 9), se intenta corregirlo. Para ello realiza el Algoritmo Berlekamp–Massey [Can11] (líneas 11 a 30), donde se calcula la discrepancia d (línea 13). Si $d = 0$ se asume que el polinomio localizador del error C es correcto. En caso contrario, se reajusta (líneas 18 a 20) y se actualizan las variables del algoritmo (líneas 21 a 28) en función del número de errores L. A continuación, se localizan los errores desde el polinomio C (líneas 31 a 33), se realiza el Algoritmo de Forney (línea 34, veáse la Sección 2.2) y se corrige el error (línea 35).

Las características técnicas del código seleccionado son, en concreto:

- Símbolos de 8 bits.
- Bloque de entrada $k = 223$ símbolos.
- Bloque de salida $n = 255$ símbolos.
- Polinomio generador del Campo de Galois $F(x) = x^8 + x^7 + x^2 + x^1 + 1$.

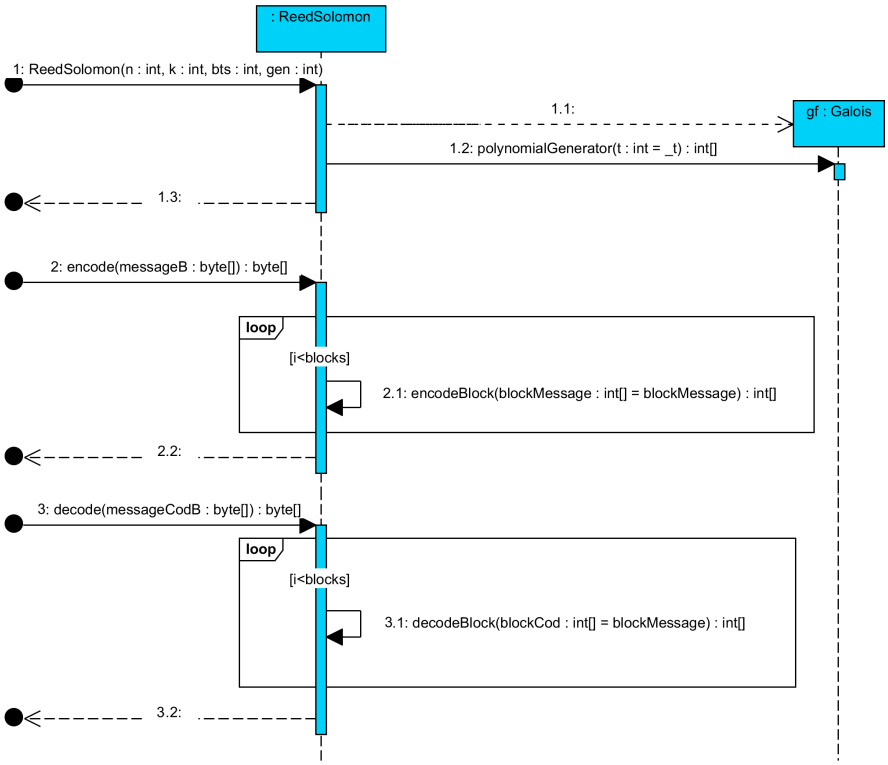

Figura 4.2: Diagrama de secuencia de Reed-Solomon.

Algoritmo 1: *decodeBlock*

Input : Block *blockCod*
Output: Block checked and corrected

```
1  for i ∈ [0 ... |2t|] do
2  |  Syndrome S = blockCod with x = α^i
3  end
4  notZero = FALSE
5  for i ∈ [0 ... |S|] do
6  |  if S_i ≠ 0 then
7  |  |  notZero = TRUE
8  |  end
9  end
10 if notZero then
11 |  B = 1; C = 1; L = 0; b = 1; m = 1;
12 |  for i ∈ [0 ... |S|] do
13 |  |  d = Σ_{j=0}^{L} S_{i−j} · C_j
14 |  |  if d = 0 then
15 |  |  |  Next m
16 |  |  else
17 |  |  |  T = C
18 |  |  |  for j ∈ [0 ... (|S| − m)] do
19 |  |  |  |  C_{m+j} = C_{m+j} + b_j
20 |  |  |  end
21 |  |  |  if 2L ≤ i then
22 |  |  |  |  L = i + 1 − L
23 |  |  |  |  B = T
24 |  |  |  |  b = d
25 |  |  |  |  m = 1
26 |  |  |  else
27 |  |  |  |  Next m
28 |  |  |  end
29 |  |  end
30 |  end
31 |  for i ∈ [0 ... n] do
32 |  |  ErrLoc = C with x = α^i
33 |  end
34 |  ErrValue = correctError(ErrLoc, S)
35 |  blockCod = blockCod + ErrValue
36 end
37 return blockCod
```

4.3. Low Density Parity Check

El diagrama de secuencia de la Figura 4.3 muestra las funciones públicas a las que se puede invocar: la instanciación y las funciones *encode()* y *decode()*. Durante la instancia del objeto se recibe el tamaño de bloque y el número máximo de iteraciones, y se generan las matrices G y H.

Estas matrices se cargan desde un fichero de texto que ha sido realizado en Matlab mediante el Algoritmo 2. Siguiendo el estándar del CCSDS [CCS11] se rellenan las submatrices de tamaño MxM, con $M = 512$, de la matriz H, de tamaño $3M$x$5M$ (líneas 1 a 5); a partir de esta matriz, se separa en dos matrices, Q, de tamaño $3M$x$2M$ (línea 6) y P, de tamaño $3M$x$3M$ (línea 7), se invierte P (línea 8) y se multiplica por Q (línea 9), la matriz resultante es la matriz W, que al trasponerla se queda de tamaño $3M$x$2M$ (línea 10); finalmente se forma la matriz G concatenando la matriz identidad de $2M$ y W (línea 11), que tendrá tamaño $2M$x$5M$, y se guardan por separado en ficheros de texto.

Algoritmo 2: *HGgenerator*

```
   Output: Generate H matrix and G matrix
 1 for i ∈ [0...2] do
 2 |   for j ∈ [0...4] do
 3 |   |   Fill Submatrix H_{i,j}
 4 |   end
 5 end
 6 Matrix Q = First 2M columns of H
 7 Matrix P = Last 3M columns of H
 8 Invert P
 9 Matrix W = P · Q
10 Traspose W
11 Matrix G = First 2M columns are 2M identity, following 3M are W
12 Save G on Gmatrix1024.txt
13 Save H on Hmatrix1024.txt
```

En las funciones *encode()* y *decode()* se divide el mensaje en bloques del tamaño especificado para que las funciones *encodeBlock()* y *decodeBlock()* operen.

En el Algoritmo 3 se explica cómo funciona la función *decodeBlock()* con mayor detalle. Para cada bloque recibido se realiza el proceso ya mostrado en la decodificación LDPC (véase la Sección 2.5): inicialización de las probabilidades (líneas 1 a 8), cálculo en VN (líneas 10 a 14), cálculo en CN (líneas 15 a 19), cálculo de probabilidad (líneas 20 a 28), computación **Hard** (líneas 29 a 33) y cálculo de syndrome (líneas 34 a 36).

Las características técnicas del código seleccionado son, en concreto:

- Bloque de entrada $k = 1024$ bits. Aunque en un principio se quería utilizar un bloque de $k = 16384$ bits, por falta de capacidad computacional este valor se

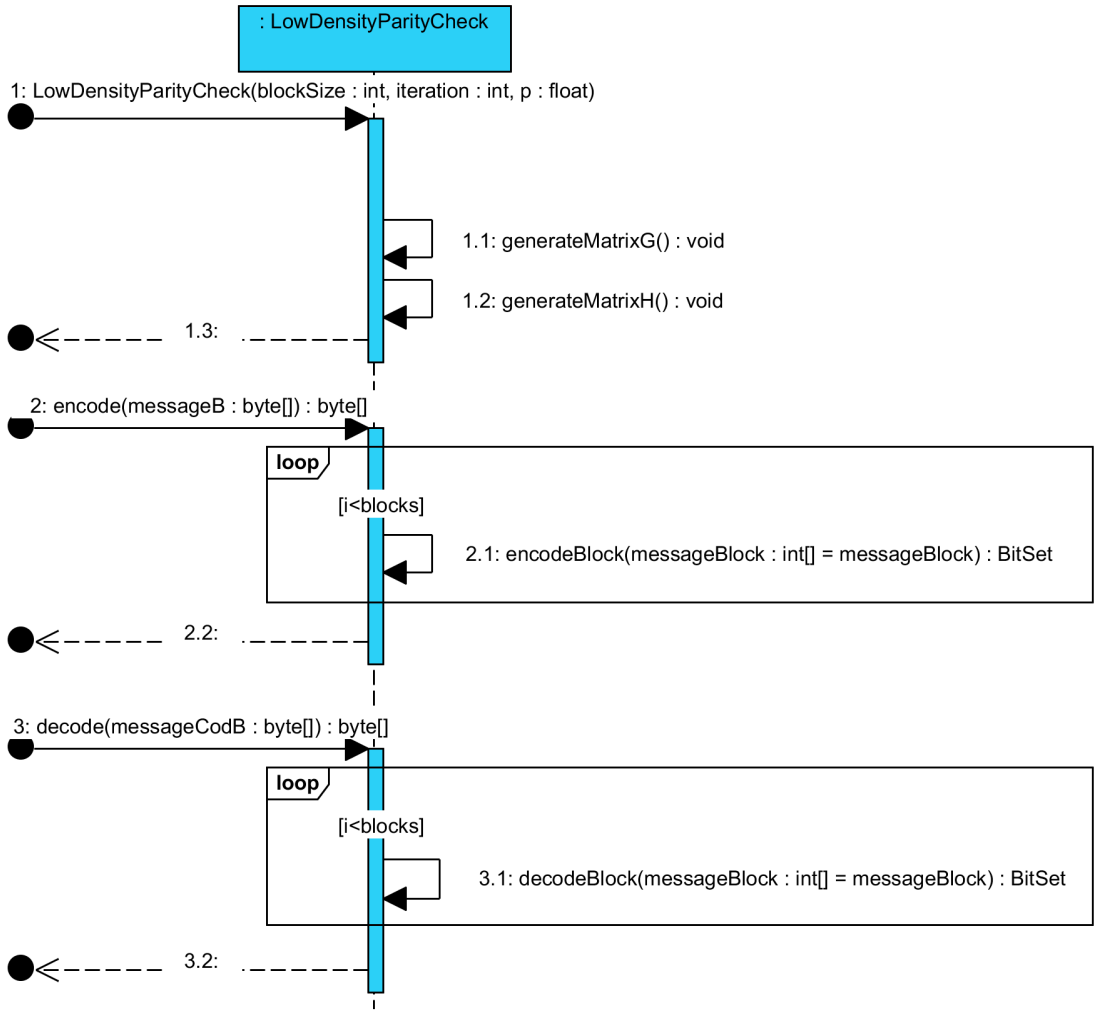

Figura 4.3: Diagrama de secuencia del LDPC.

redujo a 1024.

- Tasa de código 1/2.
- Submatrices de $M = 512$ bits.

Algoritmo 3: *decodeBlock*

Input : Block *blockCod*
Output: Check and correct the Block

for $j \in [0 \dots 5M]$ **do**
  $p_j = \frac{1}{1 + e^{2 * blockCod_j / \sigma^2}}$
  for $i \in [0 \dots 3M]$ **do**
    if $H_{i,j} = 1$ **then**
      $q_{i,j} = pj$
    end
  end
end
while $\exists\ S_j \neq 0 \wedge$ *iterations* $<$ *Max iterations* **do**
  foreach $i \in [0 \dots 3M]$, $j \in [0 \dots 5M]$ **do**
    if $H_{i,j} = 1$ **then**
      $r_{i,j} = \frac{1}{2} + \frac{1}{2} \prod_{i' \in VN_j | i' \neq j} (1 - 2q_{i',j})$
    end
  end
  foreach $i \in [0 \dots 3M]$, $j \in [0 \dots 5M]$ **do**
    if $H_{i,j} = 1$ **then**
      $q_{i,j} = p_i \prod_{j' \in CN_i | j' \neq i} r_{i,j'}$
    end
  end
  for $j \in [0 \dots 5M]$ **do**
    $found = FALSE$
    while $NOT\ found$ **do**
      if $H_{i,j} = 1$ **then**
        $Q_j = q_{i,j} \cdot r_{i,j}$
        $found = TRUE$
      end
    end
  end
  for $j \in [0 \dots 5M]$ **do**
    if $Q_j > 0{,}5$ **then**
      $blockDecod_j = 1$
    end
  end
  for $j \in [0 \dots 5M]$ **do**
    $S_j = blockDecod \cdot H_j$
  end
end
return *blockDecod*

4.4. Convolucional

El diagrama de secuencia de la Figura 4.4 muestra las funciones públicas a las que se puede invocar: la instanciación y las funciones *encode()* y *decode()*. Durante la instancia se recibe el parámetro K y los polinomios generadores de cada conexión, y se genera el Diagrama de Trellis ha utilizar en el código; en la función *encode()* se realiza la codificación ya explicada en el código convolucional (véase la Sección 2.3).

En el Algoritmo 4 se explica cómo funciona la función *decode()* con mayor detalle. Para el mensaje recibido por la función, se generará la matriz *viterbi*, en la que se guardarán los caminos con mayor métrica; y *pState*, en la que se almacena el estado anterior desde el que se accede al estado actual. Estas matrices están directamente relacionadas.

Para cada estado del Trellis (líneas 2, 3 y 4) se obtienen los posibles estados a los que puede cambiar, y se calcula la similitud entre el bloque de bits del mensaje y la salida correspondiente (líneas 6 y 7). Si el total del estado actual es mayor que el presente en la matriz *viterbi*, se actualizan los valores de la tabla y de *pState* (líneas 8 a 11). Con la matriz *viterbi* completa se realiza el procedimiento inverso para la decodificación del mensaje: se comparan los valores de la última columna de *viterbi* para hallar el estado que tiene la mayor similitud acumulada (líneas 15 a 21), y desde este estado se empieza a decodificar siguiendo el camino formado en *pState* (líneas 22 a 25). Finalmente, se devuelve el mensaje.

Las características técnicas del código seleccionado son, en concreto:

- Tasa de código 1/2.
- Longitud del código 7.
- Polinomio del primer generador $G_1(x) = x^6 + x^5 + x^4 + x^3 + 1$.
- Polinomio del segundo generador $G_2(x) = x^6 + x^4 + x^3 + x^1 + 1$.

Algoritmo 4: *decode*

Input : Message $messageCod$
Output: Check and correct the message

```
1  Matrix viterbi; Matrix pState;
2  for j ∈ [0 ... |viterbi(0,:)|] do
3      Get block_j of r bits from messageCod
4      for i ∈ [0 ... |viterbi(:,0)|] do
5          foreach k ∈ {0,1} do
6              Next State state with k as input
7              total = Compare block_j and Trellis of state
8              if viterbi_{i,j-1} + total > viterbi_{state,j} then
9                  Maximum on viterbi_{state,j} is viterbi_{i,j-1} + total
10                 Previous State pState_{state,j} is i;
11             end
12         end
13     end
14 end
15 maximum = 0;
16 for i ∈ [0 ... |viterbi(:,0)|] do
17     if viterbi_i > maximum then
18         maximum = viterbi_i
19         Maximum State state is i
20     end
21 end
22 for j ∈ reverse [0 ... |viterbi(0,:)|] do
23     Add to messageDecod input to state state, j
24     state = pState_{state,j}
25 end
26 return messageDecod
```

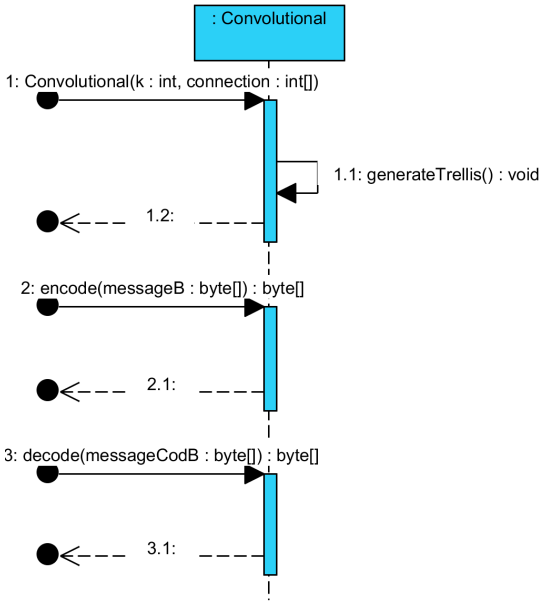

Figura 4.4: Diagrama de secuencia del código Convolucional.

4.5. Turbo Código

El diagrama de secuencia del Turbo Código de la Figura 4.5 se puede agrupar en tres bloques: instanciación, codificación y decodificación. Durante la instancia del objeto se recibe el parámetro K, el polinomio de generación y de retroalimentación del generador, el tamaño de bloque y el número de iteraciones; y se genera el Diagrama de Trellis de los codificadores convolucionales y el vector entrelazador del mensaje.

En el bloque de codificación se pueden ver las funciones *encode()* y *encodeBlock()* – aunque sólo *encode()* será accesible desde fuera del objeto. Esta función *encode()* divide el mensaje en bloques del tamaño especificado para *encodeBlock()*, que codifica el bloque sin modificar y el entrelazado, siendo necesario puntear a la mitad estos bloques para adaptarlos a la tasa 1/2.

En el bloque de decodificación se puede ver la función pública *decode()* que separa los bloques para *decodeBlock()*. Esta función inicia los dos turbo decodificadores y realiza iteraciones hasta el máximo indicado, en las que alternativamente se trabaja sobre cada decodificador.

Durante la instancia de la clase **TurboDecoder** se reciben: los bits de información, Y_s; los bits de paridad correspondiente al codificador, Y_p; y el Trellis del convolucional del sistema, Tr. En el Algoritmo 5 se explica cómo funciona la decodificación del **TurboDecoder**.

Siguiendo lo especificado en la decodificación del Turbo Código (véase la Sección 2.4), se calcula $\gamma(s', s)$ (líneas 1 a 4), $\alpha(s)$ (líneas 5 a 9) y $\beta(s)$ (líneas 10 a 20). Para la inicialización de valores, se considera $\alpha_{0,0} = 1$ ya que siempre empieza el Diagrama

de Trellis desde el estado 0; y, si el flujo no ha sido entrelazado (primer decodificador), $\beta_{0,K} = 1$, ya que acaba en el estado 0; en caso contrario, $\beta_{s,K}$ será equiprobable en todos los estados. Finalmente, se calcula la estimación MAP (líneas 21 a 29).

Las características técnicas del código seleccionado son, en concreto:

- Bloque de entrada $k = 8920$ bits.
- Tasa de código 1/2.
- Profundidad del entrelazado 5. Viene dado por el tamaño del bloque de entrada.
- Longitud del código Convolucional 5.
- Polinomio de retroalimentación del generador $G_0(x) = x^4 + x^1 + 1$.
- Polinomio de generador $G_1(x) = x^4 + x^3 + x^1 + 1$.

Algoritmo 5: *TurboDecoder*

Input : Extrinsic Information Le
Output: Estimate the original message

foreach $i \in [0 \dots 2^K]$; $j \in [0 \dots |Ys|]$ **do**
  $c1 = i \gg (K-1)$
  $\gamma_{i,j} = \exp[\frac{1}{2}(c1 \cdot (Le_j + Lc \cdot Ys_j) + Lc \cdot Yp_j \cdot Tr_i)]$
end
Initialize $\alpha_{0,0} = 1$
foreach $j \in [1 \dots |Ys|]$, $i \in [0 \dots 2^{K-1}]$, $k \in \{0, 1\}$ **do**
  Calculate next *state*
  $\alpha_{state}(j) += \alpha_i(j-1) \cdot \gamma_{state,j-1}$
end
if *is* $Decoder_1$ **then**
  Initialize $\beta_{0,Lastcolumn} = 1$
else
  foreach $i \in [0 \dots 2^{K-1}]$ **do**
    Initialize $\beta_{i,Lastcolumn} = \frac{1}{2^{K-1}}$
  end
end
foreach $j \in$ *reverse* $[1 \dots |Ys|]$, $i \in [0 \dots 2^{K-1}]$, $k \in \{0, 1\}$ **do**
  Calculate previous *state*
  $\beta_i(j) += \beta_{state}(j+1) \cdot \gamma_{state,j+1}$
end
foreach $j \in [0 \dots |Ys|]$, $i \in [0 \dots 2^K]$ **do**
  $i_\alpha = i \wedge (2^{K-1} - 1)$
  $\sigma_{i,j} = \alpha_{i_\alpha,j} \cdot \gamma_{i,j} \cdot \beta_{Tr_i,j}$
end
for $j \in [0 \dots |Ys|]$ **do**
  Add $minus_j$ states
  Add $plus_j$ states
  Estimation $E_j = \log \frac{plus_j}{minus_j}$
end
return E

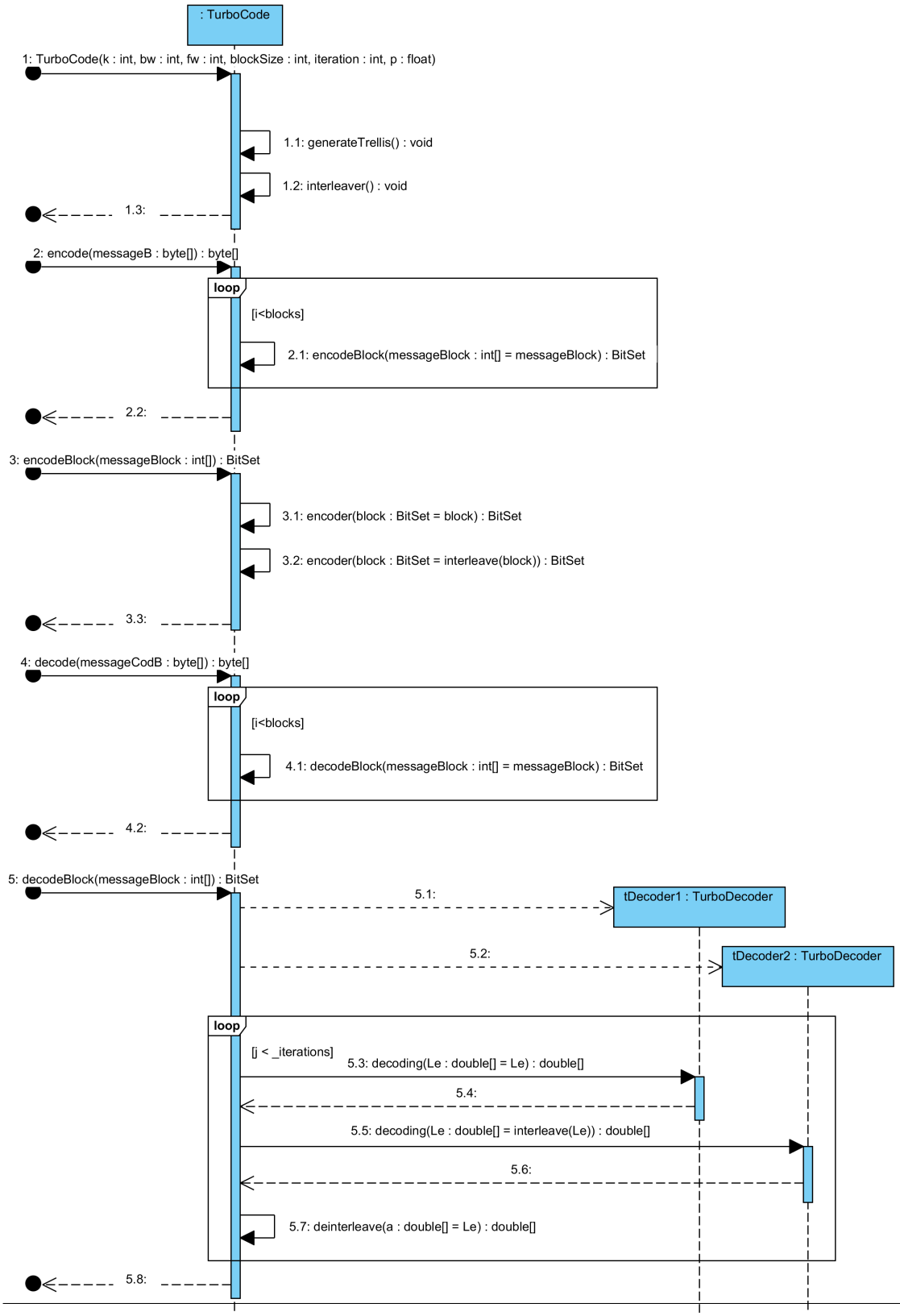

Figura 4.5: Diagrama de secuencia del Turbo Código.

4.6. Disponibilidad

La documentación en formato HTML del código y el código fuente de la implementación de las técnicas FEC se encuentra disponible en el repositorio:

`https://bitbucket.org/ssarasa/fectechniques`

El código ha sido liberado bajo la licencia GNU General Public License v3.

Capítulo 5

Evaluación y resultados

Para evaluar las prestaciones de las técnicas FEC implementadas se han considerado diferentes tipos de configuraciones del sistema, limitadas por el *hardware* del equipo donde se han realizado. El equipo donde se han realizado las pruebas cuenta con un Intel Core i7-4790 CPU @ 3.60GHz x 4 y 4GBytes de memoria RAM, con el sistema operativo Debian Linux 4.2.6-1-pve ejecutándose sobre una arquitectura x64.

Para ello, se ha realizado un mismo escenario para cada código (véase la Figura 5.1), donde se pueden variar hasta tres parámetros:

- El fichero enviado como mensaje, que puede ser de tamaño pequeño (0.1 KB, 0.5 KB y 1 KB) o de tamaño medio (100 KB, 150 KB y 200 KB).
- La probabilidad de error durante la transmisión, que varía de 0 % a 20 %.
- En el caso de los códigos LDPC y TC, el número de iteraciones del decodificador, de valor 1, 5, 10, 15, 20, 25 y 30 iteraciones.

Para la simulación se ha realizado un *script* que varía todas las posibles configuraciones y guarda los resultados. Para cada una de las configuraciones se han realizado 10 pruebas.

En una ejecución del escenario se destacan tres fases: (i) codificación, (ii) transmisión y (iii) corrección de errores. Durante la transmisión de las comunicaciones en espacio profundo, el error que se produce es el ruido blanco Gausiano (*Additive White Gaussian*

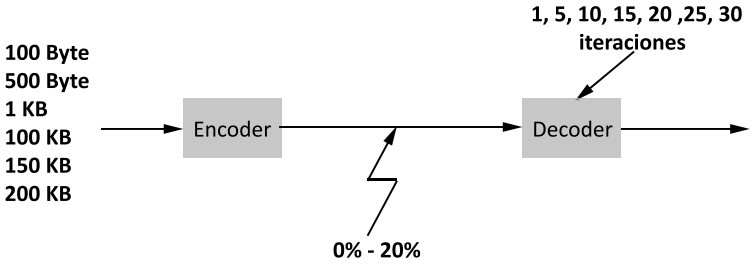

Figura 5.1: Escenario experimentación.

Noise, AWGN)[PB00]. Para la inserción de este error durante transmisión, se han simulado dos tipos de ruido, en función del decodificador: ruido *hard*, que invierte el bit (este tipo de ruido se aplica a los códigos RS y convolucional); y ruido *soft*, que se añade a la transmisión (aplicado a los códigos LDPC y TC).

Para la evaluación de rendimiento se han comparado los resultados en función del tamaño de la transmisión, tiempo y corrección de error.

Respecto al tamaño de la codificación, el RS tiene $r = 223/255$, mientras que el resto de códigos tienen $r = 1/2$, por lo tanto el tamaño teórico del mensaje codificado será el doble.

En la Tabla 5.1 se puede ver el tamaño experimental de los mensajes codificados. Las discrepancias entre el tamaño teórico y el tamaño experimental se deben a que, salvo el código convolucional, el resto de códigos necesitan dividir primero el mensaje en bloques. Por lo tanto, si un bloque no llega a completarse, se rellena con ceros. El código que añade mayor sobrecarga en transmisión mientras el mensaje es de tamaño pequeño es el Turbo Código (ya que tiene un tamaño de bloque de 8920 bits), ajustándose al resto de $r = 1/2$ conforme aumenta el tamaño del mensaje.

Tamaño fichero	RS	%	CONV	%	TC	%	LDPC	%
100 Byte	132 Byte	1.32	200 Byte	2	2231 Byte	22.31	256 Byte	2.56
500 Byte	596 Byte	1.192	1000 Byte	2	2231 Byte	4.462	1024 Byte	2.048
1000 Byte	1160 Byte	1.16	2000 Byte	2	2231 Byte	2.231	2048 Byte	2.048
100 KB	114,368 KB	1.144	200 KB	2	200,79 KB	2.007	200,192 KB	2.002
150 KB	171,536 KB	1.144	300 KB	2	301,185 KB	2.008	300,032 KB	2
200 KB	228,704 KB	1.144	400 KB	2	401,58 KB	2.008	400,128 KB	2

Tabla 5.1: Tabla comparación de tamaño codificado.

Para las comparaciones de tiempo y corrección de error se han limitado los ficheros a dos, el pequeño de 1 KB y el medio de 150 KB, ya que comparando los resultados entre los otros ficheros se ha comprobado que el tiempo de operación es proporcional al tamaño y que la capacidad de corrección no varía según el tamaño.

En la Figura 5.2 se muestra un diagrama de cajas de comparación de tiempos de codificación. Este diagrama permite mostrar los máximos, mínimos y cuartiles de distribución de un conjunto de datos. El diagrama de cajas de la Figura 5.2 muestra para el fichero de 1 KB unos tiempos de codificación cercanos a cero, con algún valor atípico no mayor de 12 milisegundos para las implementaciones del RS, convolucional y TC; para la codificación del LDPC se puede apreciar un valor mediano de 125 ms y un valor típico entre 110 y 140 ms.

En la comparación de tiempos de codificación y decodificación, véase la Figura 5.2 y la Figura 5.3 respectivamente, se puede ver que el código que utiliza más tiempo en ambos procesos es el LDPC. Para no perder la perspectiva en la comparación, se ha eliminado en ambas situaciones el tiempo del proceso para el código LDPC con el fichero medio, ya que en ambos casos cambia la magnitud de la medida temporal en 2 órdenes de magnitud.

En la Figura 5.4, se muestra la comparativa del tiempo de decodificación del código

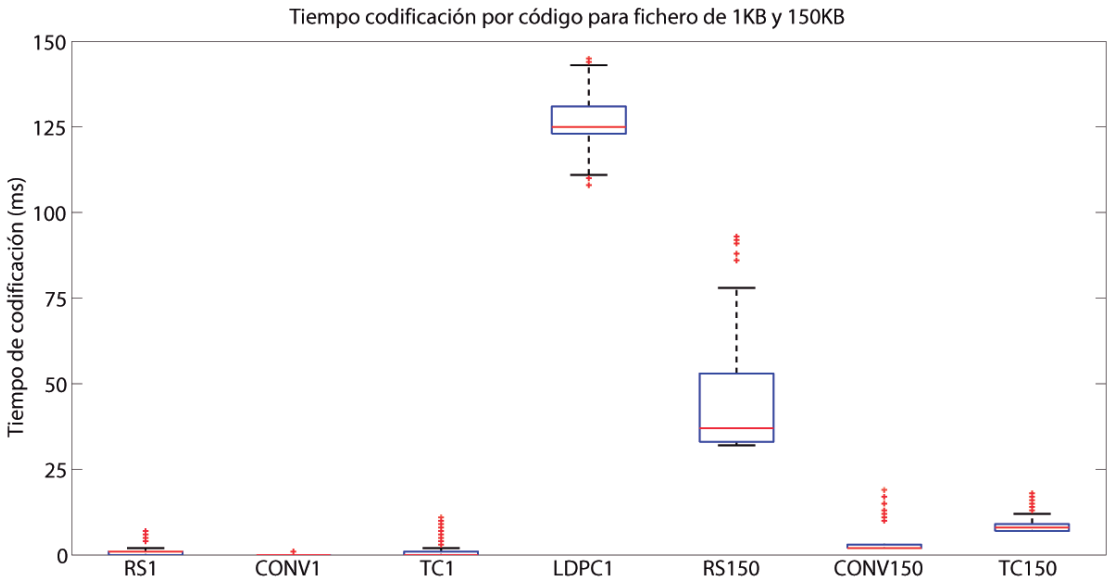

Figura 5.2: Comparativa tiempos de codificación por código.

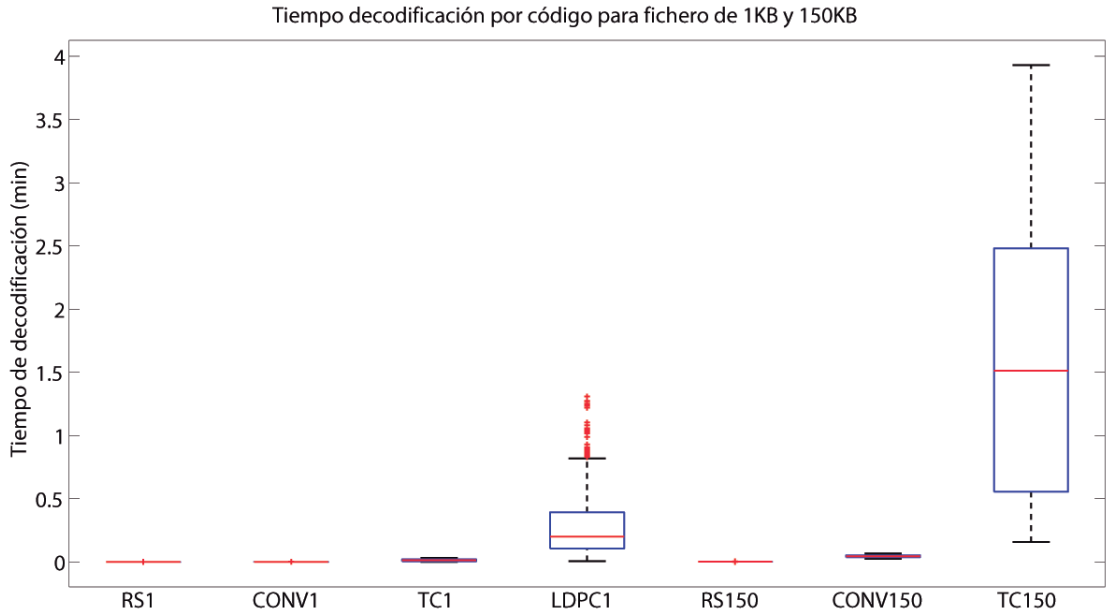

Figura 5.3: Comparativa tiempos de decodificación por código.

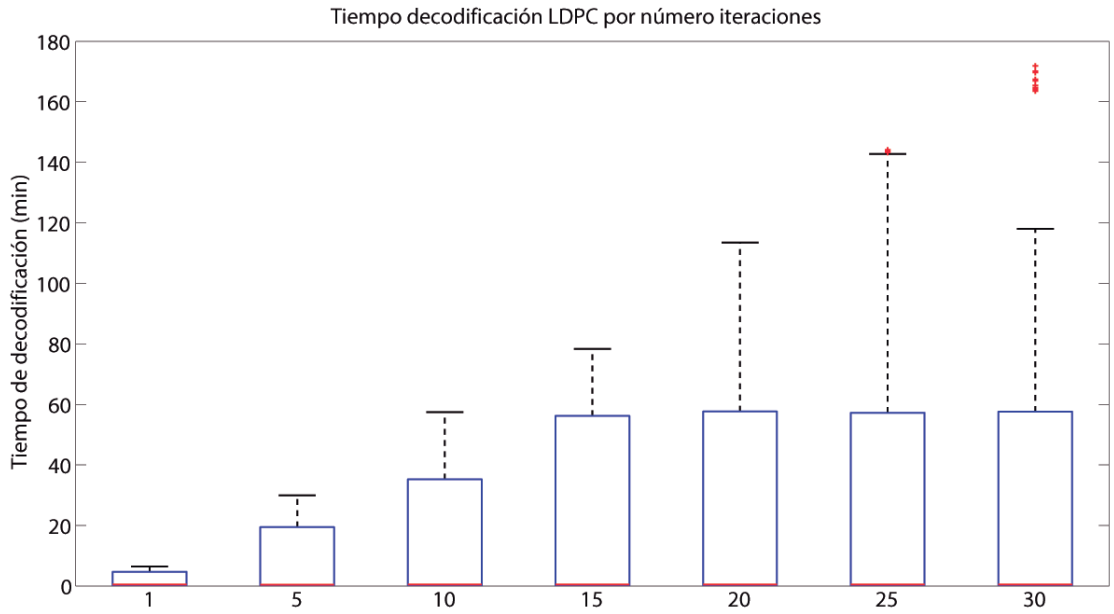

Figura 5.4: Comparativa tiempos de decodificación según número iteraciones.

LDPC con el fichero de 150 KB según el número de iteraciones utilizado. Según lo explicado en la decodificación LDPC (Sección 2.3), se puede ver que a partir de las 15 iteraciones se estabiliza al conseguir corregir el error, y por tanto, no necesita acabar con las iteraciones restantes. Los casos atípicos presentes son los casos en los que ha realizado todas las iteraciones sin corregir el error.

En la Figura 5.5, se muestra la comparativa entre FEC de los errores al final del escenario frente al error insertado para el fichero de 150 KB. Se puede observar el comportamiento descrito en la Sección 2.2 de manera experimental. El código RS solo es capaz de corregir 16 símbolos del bloque de 255 (es decir, hasta un 6.27 % de error), por lo tanto, no es capaz de recuperar demasiados errores, y eso se aprecia en la relación prácticamente proporcional de la gráfica. El código convolucional, ante la inserción de muchos errores no es capaz de realizar una corrección coherente, llegando incluso a añadir más errores de los presentes. Sin embargo, cuando el error es menor de 11 %, produce mejores resultados que el RS, incluso consiguiendo corregir por completo el mensaje si el error es menor de 6 %. El TC es el código con mejores resultado de la experimentación, corrigiendo totalmente el mensaje si el error introducido es menor de 13 %.

La línea gráfica del LDPC en la Figura 5.5 muestra un resultado muy irregular debido a que la diferencia de iteraciones del código produce correcciones muy diferentes. Por ello, en la Figura 5.6 se muestra una comparativa más detallada de los resultados de la corrección del código LDPC. Como se observa, las correcciones a partir de las 15 iteraciones producen resultados muy similares; en concreto, 20, 25 y 30 iteraciones corrigen hasta 11 % de error insertado, y 15 iteraciones hasta 9 %.

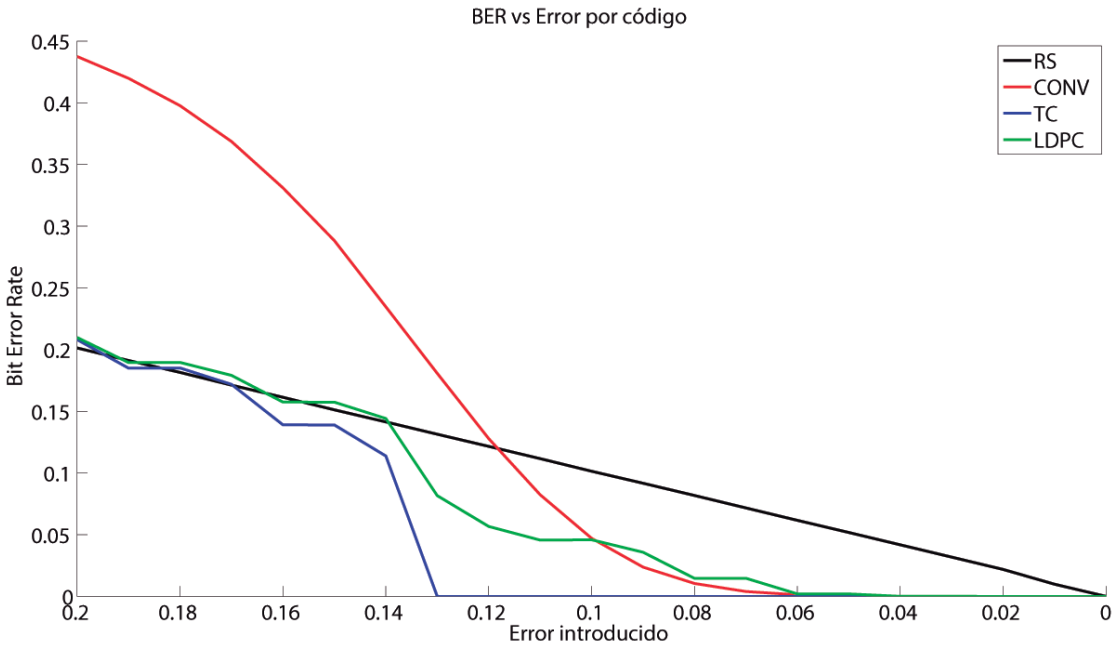

Figura 5.5: Comparativa BER vs Error introducido por código.

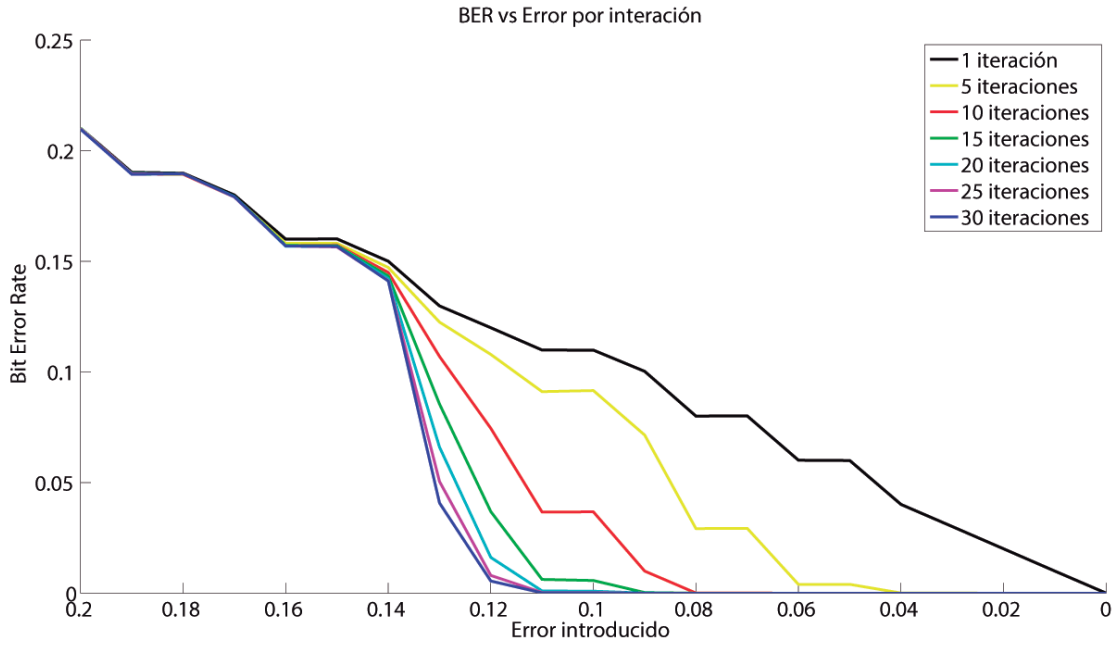

Figura 5.6: Comparativa BER vs Error introducido por iteración.

En conclusión, los valores experimentales indican que en comparación al resto de técnicas, el Turbo Código es el código con una mayor capacidad correctora, llegando incluso hasta la corrección total con tasas de error de bit aleatorio inferiores al 13 %. Los procesos de codificación y decodificación tardan un tiempo medio del orden de 25 milisegundos y 2 minutos, respectivamente. Sin embargo, presenta una sobrecarga de bits mayor que los demás, pero que se compensa cuanto mayor es el tamaño del mensaje. Otra desventaja es que el proceso de decodificación es un proceso computacionalmente complejo, ya que los diagramas que utiliza son muy elaborados y además no posee un proceso intermedio de comprobación del mensaje para terminar antes de realizar todas las iteraciones si el mensaje ya está corregido.

Capítulo 6

Conclusiones

Este capítulo presenta algunas conclusiones obtenidas de la elaboración de este TFG así como posibles líneas de trabajo futuras con los que mejorarlo.

La teoría de codificación es utilizada desde que en 1948 Claude Shannon publicó *A Mathematical Theory of Communication* [Sha48], donde se enfoca en cómo conseguir codificar la información del emisor en una comunicación. Desde ese momento se han desarrollado muchas técnicas para conseguir aproximarse al Límite de Shannon, con diferentes ventajas entre ellas. En este proyecto se pueden apreciar algunas de estas técnicas y cómo se comportan frente a errores.

Como resultado del proyecto se ha implementado un conjunto de técnicas funcionales que se pueden aplicar en comunicaciones con error de bit aleatorio. Además, se han adquirido conocimientos sobre la teoría matemática de estas técnicas. Pese a los inconvenientes debido a las limitaciones de *hardware* y *software*, se ha probado la viabilidad y funcionamiento de las técnicas implementadas. Mediante simulaciones, además, se ha comprobado su rendimiento por separado y comparando entre ellas, destacando el Turbo Código como la más eficiente en la corrección de error.

Las líneas de trabajo futuros se pueden dividir en función de los ámbitos de este proyecto. En primer lugar, se pueden optimizar las implementaciones; además, se pueden mejorar algunas de las técnicas, como por ejemplo evitar iteraciones innecesarias durante la corrección del mensaje con el Turbo Código. También se pueden implementar nuevas técnicas no seleccionadas para la formación de una librería de técnicas de codificación. Por último, se puede profundizar en mecanismos de detección en la recepción para identificar la técnica aplicada y realizar la decodificación correspondiente.

Bibliografía

[BCJR74] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate (Corresp.). *IEEE Transactions on Information Theory*, 20(2):284–287, Mar 1974.

[BDMP96] S Benedetto, D Divsalar, G Montorsi, and F Pollara. A soft-input soft-output maximum a posteriori (MAP) module to decode parallel and serial concatenated codes. *The Telecommunications and Data Acquisition Progress Report*, 42(127):1–20, 1996. NASA Code 315-91-20-20-53.

[Can11] Anne Canteaut. Berlekamp–Massey Algorithm. In *Encyclopedia of Cryptography and Security*, pages 80–80. Springer, 2011.

[CCS11] CCSDS. TM Synchronization and Channel Coding. [Online], August 2011. `http://public.ccsds.org/publications/archive/131x0b2ec1.pdf`.

[CCS12] CCSDS. TM Synchronization and Channel Coding—Summary of Concept and Rationale. [Online], November 2012. `http://public.ccsds.org/publications/archive/130x1g2.pdf`.

[For65] G David Forney. On decoding BCH codes. *Information Theory, IEEE Transactions on*, 11(4):549–557, 1965.

[ISR60] G. Solomon I. S. Reed. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[Kol13] Michael Olorunfunmi Kolawole. *Satellite Communication Engineering, 2nd Edition*. CRC Press, 2013.

[Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, Sept 1955.

[MG10] Todd K. Moon and Jacob H. Gunther. AN INTRODUCTION TO LOW-DENSITY PARITY-CHECK CODES. In *Proceedings of the International Telemetering Conference*, volume 39. International Foundation for Telemetering, October 2010.

[MN96] David JC MacKay and Radford M Neal. Near Shannon limit performance of low density parity check codes. *Electronics letters*, 32(18):1645–1646, 1996.

[MSM14] Salehe I Mrutu, Anael Sam, and Nerey H Mvungi. Forward Error Correction Convolutional Codes for RTAs' Networks: An Overview. *International Journal of Computer Network and Information Security*, 6(7):19, 2014.

[PB00] David R Pauluzzi and Norman C Beaulieu. A comparison of SNR estimation techniques for the AWGN channel. *IEEE Transactions on Communications*, 48(10):1681–1691, 2000.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[RSU01] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke. Design of capacity-approaching irregular low-density parity-check codes. *IEEE Transactions on Information Theory*, 47(2):619–637, Feb 2001.

[Sha48] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[Skl97] Bernard Sklar. Turbo code concepts made easy, or how I learned to concatenate and reiterate. In *MILCOM 97 Proceedings*, volume 1, pages 20–26. IEEE, 1997.

[Uni10a] CS Duke University. Decoding Reed-Solomon Codes. [Online], 2010. `http://www.cs.duke.edu/courses/spring10/cps296.3/decoding_rs_scribe.pdf`.

[Uni10b] CS Duke University. Reed-Solomon Codes. [Online], 2010. `https://www.cs.duke.edu/courses/spring10/cps296.3/rs_scribe.pdf`.

[VA10] VA. Turbo Code Primer. [Online], 2010. `http://vashe.org/turbo/turbo_primer_0.0.pdf`.

[Vit67] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[ZZL13] Peng Zhu, Jun Zhu, and Xiang Liu. A Study on Turbo Code Performance Based on AWGN Channel. In *Applied Mechanics and Materials*, volume 347, pages 1720–1726. Trans Tech Publ, 2013.

Acrónimos

- **FEC** Forward Error Correction
- **CCSDS** Consultative Committee for Space Data System
- **RS** Reed-Solomon
- **GF** Galois Field
- **RTA** Real Time Application
- **AWGN** Additive White Gaussian Noise
- **LDPC** Low Density Parity Check
- **ECC** Error-Correcting Code
- **UML** Unified Modeling Language
- **ARQ** Automatic Repeat Request
- **LFSR** Lineal Feedback Shift Register
- **ISO** International Organization for Standardization
- **NASA** National Aeronautics and Space Administration

Anexo A

Horas de trabajo

Con el objetivo de controlar el tiempo de desarrollo del proyecto, se ha realizado un seguimiento de los días dedicados a cada parte del mismo. En la figura A.1 se puede ver el diagrama de Gantt por semanas y tareas.

La primera etapa del proyecto consistió en estudiar qué es un ECC, cuáles son sus tipos y qué son las técnicas FEC. Además, se seleccionaron las técnicas a implementar. Tras el análisis del problema y el estudio de las características de la arquitectura *software*, se implementaron las técnicas seleccionadas, apoyándose en el estudio de cada una. Finalmente, se realizaron las pruebas y se evaluó el resultado.

Al final, de los 11 meses de proyecto se ha estimado un coste aproximado de 390 horas de trabajo, desglosado en la Tabla A.1. Aunque el número de horas excede las previstas para el proyecto, este tiempo ha sido necesario para la implementación de las técnicas y el tiempo de pruebas.

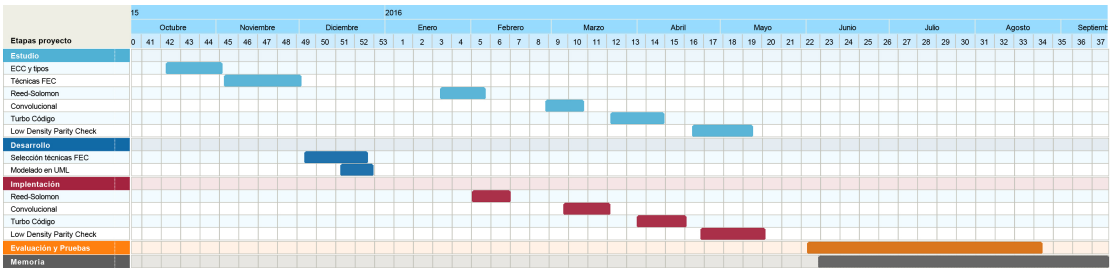

Figura A.1: Diagrama de Gantt.

Tarea	Comienzo	Fin	Duración
Estudio	13/10/2015	12/05/2016	**103h**
ECC y tipos	13/10/2015	02/11/2015	21h
Técnicas FEC	03/11/2015	01/12/2015	15h
Reed-Solomon	20/01/2016	05/02/2016	8h
Convolucional	27/02/2016	12/03/2016	14h
Turbo Código	22/03/2016	10/04/2016	19h
Low Density Parity Check	20/04/2016	12/05/2016	26h
Desarrollo	02/12/2015	27/12/2015	**45h**
Selección técnicas FEC	02/12/2015	25/12/2015	30h
Modelado en UML	15/12/2015	27/12/2015	15h
Implementación	01/02/2016	16/05/2016	**130h**
Reed-Solomon	01/02/2016	14/02/2016	24h
Convolucional	05/03/2016	21/03/2016	18h
Turbo Código	01/04/2016	18/04/2016	46h
Low Density Parity Check	23/04/2016	16/05/2016	42h
Evaluación y pruebas	01/06/2016	24/08/2016	**84d**
Memoria	05/06/2015	17/09/2016	**112h**

Tabla A.1: Duración de cada tarea.