UNIVERIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

IMPLEMENTACIÓN DE ESTRATEGIAS DE OPTIMIZACIÓN EN UN BANCO DE PRUEBAS PARA REDES INALÁMBRICAS DE SENSORES COGNITIVAS

Manuel Alarcón Granero

Julio 2015

TRABAJO FIN DE GRADO

Título:	IMPLEMENTACIÓN DE ESTRATEGIAS DE OPTIMIZACIÓN EN UN BANCO DE PRUEBAS PARA REDES INALÁMBRICAS DE SENSORES COGNITIVAS
Autor:	MANUEL ALARCÓN GRANERO
Tutor:	ELENA ROMERO PERALES
Ponente:	ALVARO ARAÚJO PINTO
Departamento:	DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

TRIBUNAL

Presidente:	D. OCTAVIO NIETO-TALADRIZ GARCÍA
Vocal:	D. ÁLVARO DE GUZMÁN FERNÁNDEZ GONZÁLEZ
Secretaría:	D. ALVARO ARAÚJO PINTO
Suplente:	D. MIGUEL ÁNGEL SÁNCHEZ GARCÍA

CALIFICACIÓN:

Madrid, a de de 2015.

UNIVERIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

IMPLEMENTACIÓN DE ESTRATEGIAS DE OPTIMIZACIÓN EN UN BANCO DE PRUEBAS PARA REDES INALÁMBRICAS DE SENSORES COGNITIVAS

Manuel Alarcón Granero

Julio 2015

Resumen

Abstract

Índice

LISTA DE FIGURAS

Capítulo 1. Introducción

En este capítulo se van a llevar a cabo una introducción al trabajo en la que se describirán los objetivos principales del trabajo y se detallará el proceso para desarrollarlo. Por último, se describirá la estructura de esta memoria.

1.1. Objetivos

Aksdlfh

1.2. Desarrollo del trabajo

El trabajo se ha dividido en las siguientes etapas:

- Estudio previo. El primer paso tomado para la consecución de este trabajo ha sido la adquisición de conocimientos sobre las redes de sensores inalámbricas cognitivas y con el entorno de trabajo.
    - Estudio de trabajos realizados anteriormente sobre CWSN en el laboratorio y de la documentación facilitada sobre los algoritmos que se han implementado.
    - Familiarización con las herramientas de programación de Microchip, en concreto MPLAB X, y revisión de los conceptos de programación de microcontroladores en C.
- Desarrollo. Consiste en la implementación de los dos algoritmos sobre los nodos disponibles en el laboratorio. Tras la implementación de los dos algoritmos se ha desarrollado una aplicación que sirva de demostración del correcto funcionamiento del código implementado.
- Pruebas. Se han realizado las pruebas necesarias para comprobar el comportamiento de los algoritmos en el nodo. Por ejemplo, introduciendo una fuente de ruido cerca de un nodo para comprobar que se inicia el algoritmo de reducción de consumo.
- Documentación. La escritura de esta memoria ha sido simultánea a las etapas anteriores y tras terminar la implementación de cada algoritmo.

1.3. Estructura de la memoria

Capítulo 2. Redes de sensores inalámbricas cognitivas

En este capítulo se resumirán las principales características de las redes de sensores inalámbricas (WSN, en sus siglas en inglés) para, a continuación, introducir una evolución de éstas denominada CWSN (Cognitive Wireless Sensor Networks). Por último, se presentarán algunos de los nodos para CWSN que existen en la actualidad y sus principales características.

2.1. Redes de sensores inalámbricas

Una red de sensores es aquella formada por una serie de dispositivos con acceso a información del medio cuya misión es la de monitorizar diferentes parámetros del entorno. Las WSN están formadas por dispositivos con conectividad inalámbrica, lo que les da mayor versatilidad y flexibilidad. El número de nodos que forman una red de este tipo puede variar desde unos pocos hasta varios cientos y pueden conectarse siguiendo diferentes topologías como podemos ver en la Figura 2.1.

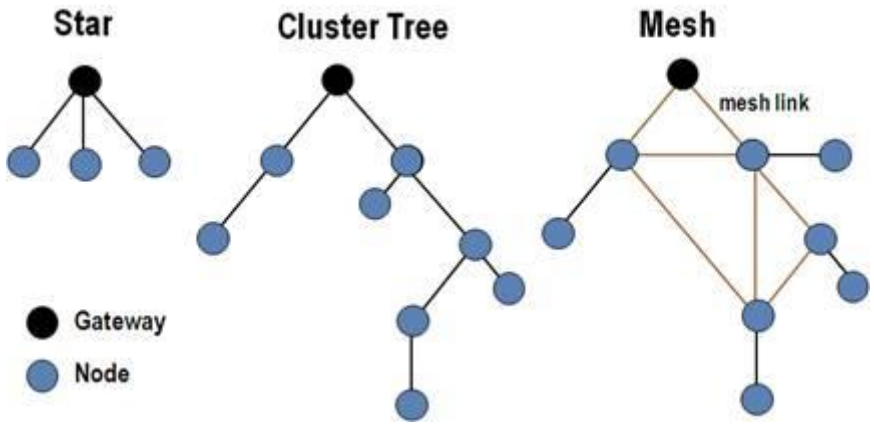


Figura 2.1 Topologías de red WSN, obtenida de [1]

En todas estas topologías de red existe un nodo, con más recursos que el resto, que hace de puerta de enlace o de coordinador para el resto de nodos, como es el caso de las redes *mesh*.

En cuanto a las diferentes tecnologías y protocolos de comunicación implementados en WSN la mayoría están basados en el estándar 802.15.4 del IEEE (*Institute of Electrical and Electronics Engineers*) [2] para WPAN (Wireless Personal Area Network). Uno de los protocolos basados en este estándar es MiWi™ [refMiWi] y es el que incorpora el nodo con el que vamos a trabajar.

Otro estándar muy utilizado es el IEEE 802.11 [ref802.11] en el que está basado Wi-Fi [refWiFi]. Debido a la extensión en el uso de Wi-Fi, otro trabajo que se está desarrollando en el LSI está dando conectividad mediante este estándar a los nodos.

2.2. Redes cognitivas

2.3. Redes de sensores inalámbricas cognitivas

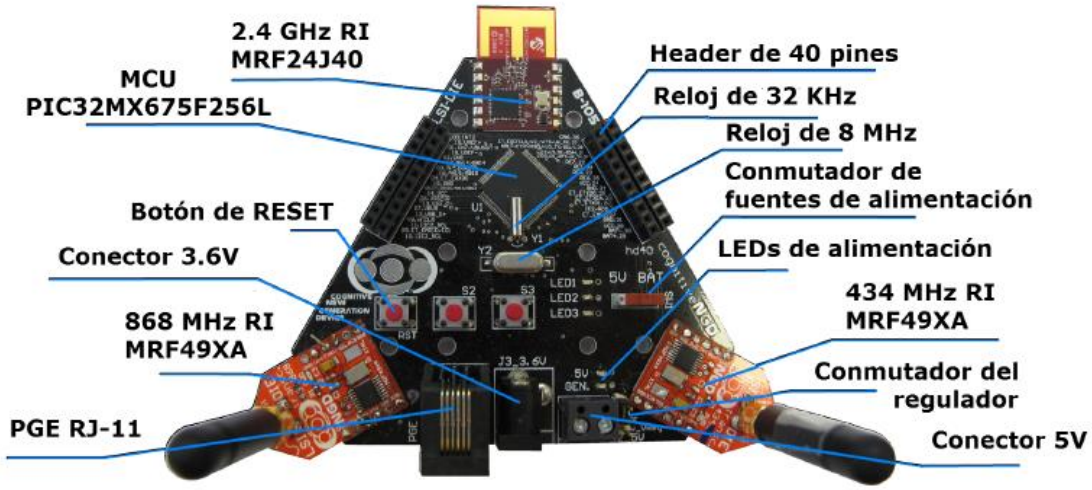
2.3.1. Nodos para CWSN

Capítulo 3. Estudio previo

En este capítulo se va a presentar la plataforma sobre la que se ha realizado el trabajo. Se detallarán tanto el hardware como el firmware del que disponen los nodos y la arquitectura cognitiva donde se ha desarrollado este trabajo. Por último, se presentarán los algoritmos que han sido el objetivo de este trabajo.

3.1. Hardware del cNGD

El hardware sobre el que se ha desarrollado en este trabajo es el nodo cNGD desarrollado en el LSI y el cual viene detallado en [1]. Aquí expondremos brevemente algunas de sus características más importantes y que se han tenido en cuenta a la hora de realizar el trabajo ya que influyen a la hora de implementar código sobre el nodo.



(3.1) Vista superior.

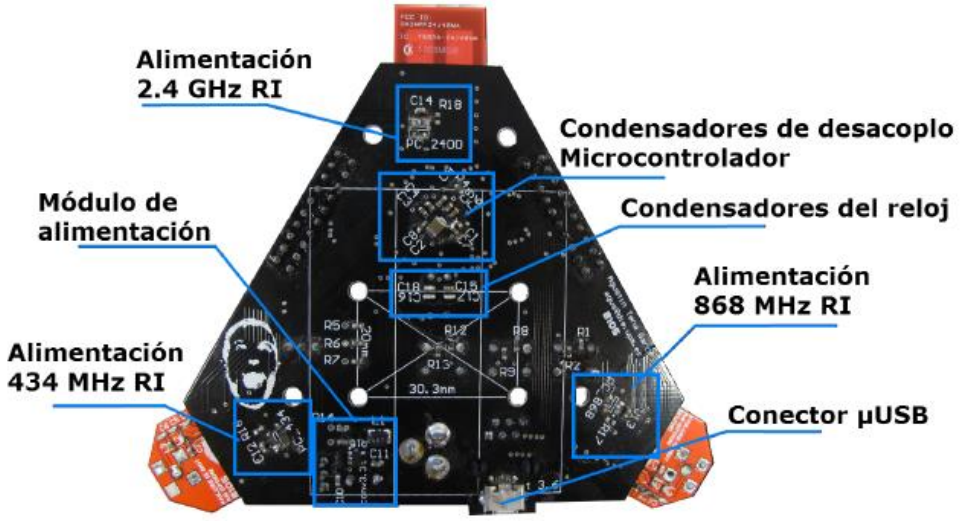


Figura 3.1 Vista detallada del cNGD.

El hardware del nodo tiene que cumplir unos requisitos necesarios en cuanto a consumo, bajos recursos, bajo coste y varias bandas de frecuencias para las comunicaciones. Por tanto pasamos a describir algunas de sus características principales:

- Microcontrolador. El MCU que incorpora el nodo es el PIC32MX675F256L [2], de 32 bits y fabricado por Microchip. Tiene 100 pines y sus características son:
    - Memoria. 256 kB de memoria flash y 64 kB de memoria RAM.
    - Reloj interno. Frecuencia máxima de funcionamiento de 80 MHz.
    - Modos de funcionamiento. Varios modos para reducir el consumo.
    - Timers. Cinco timers de 16 bits, pudiendo utilizar dos de ellos para hacer uno de 32 bits.
- Interfaces radio. El nodo dispone de tres interfaces radio, por lo que es capaz de transmitir y recibir a través de tres frecuencias diferentes. Éstas son 434 MHz, 868 MHz y 2,4 GHz. Con esto cubre todas las bandas ISM de Europa. Para poder reducir el consumo de los nodos sin tener que renunciar a tener las tres interfaces, éstas se pueden activar o desactivar cuando no se estén utilizando.
- Alimentación. El nodo tiene varias posibilidades de alimentación, siendo la principal las baterías. También se puede alimentar a través de USB, del conector RJ-11 o del conector de 3.6 V.
- Módulos de expansión. El nodo tiene la posibilidad de expandir su funcionalidad mediante módulos de expansión que se conectan a los *headers* disponibles. Uno de los que se han utilizado en este trabajo es el que permite comunicarse a través de línea serie RS232, permitiendo comprobar la funcionalidad del código.

Como vemos, las características del cNGD satisfacen las especificaciones necesarias para un nodo para CWSN, ya que es capaz de trabajar en diferentes bandas de frecuencia, en este caso todas las bandas ISM de Europa, es capaz de trabajar en modos de bajo consumo tanto con los diferentes modos de funcionamiento del microcontrolador como apagando las interfaces radio que no utilice. Además, permite el desarrollo de nuevas funcionalidades mediante los módulos de expansión.

3.2. Firmware del cNGD

El firmware implementado en el nodo y que fue desarrollado en [3] tiene la función de optimizar, adaptar e integrar las pilas de protocolos de cada uno de los transceptores en una única pila, y de proporcionar una interfaz que simplifica el trabajo del programador mediante una serie de funciones que son las que acceden al hardware.

La pila de protocolos de los transceptores se resume en la Figura 3.2, obtenida de [3], donde se ve que entre la aplicación y el transceptor sólo se pueden comunicar mediante paquetes, registros y mensajes.

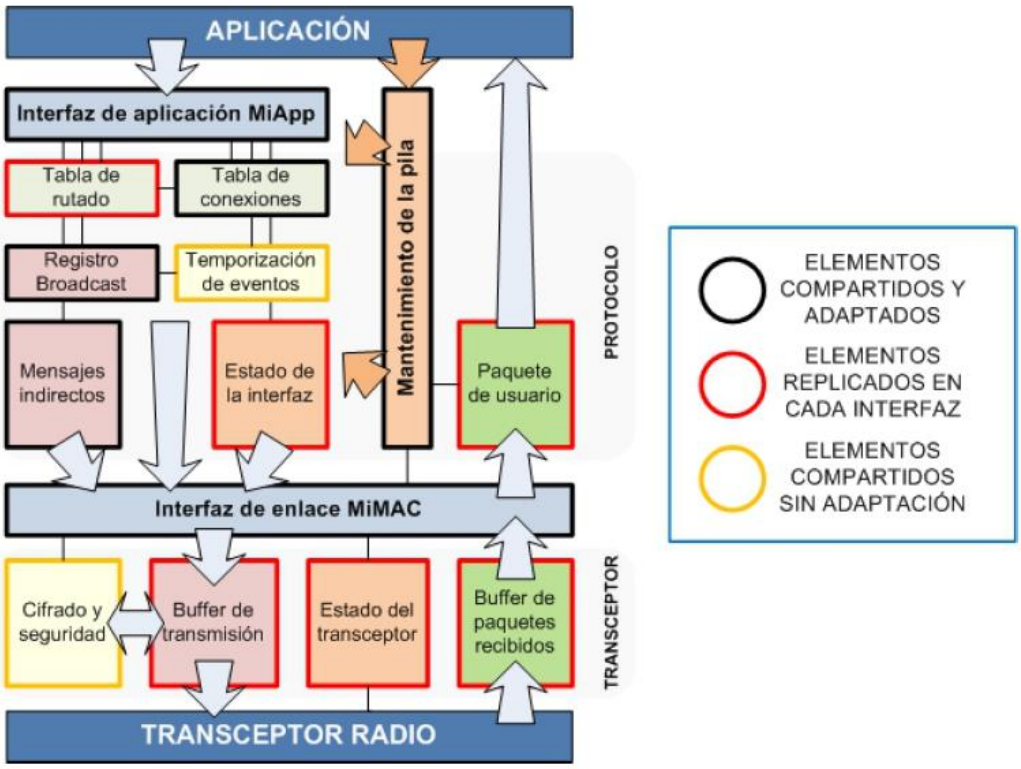


Figura 3.2

Con la adaptación de [3], que denominamos HAL, lo que se consigue es simplificar la labor del programador a la hora de utilizar la pila de protocolos de las interfaces, teniendo que hacer llamadas a funciones para acceder a los recursos y, además, la HAL da flexibilidad para cambiar los transceptores, añadir nuevos o implementar nuevas funcionalidades. Todo esto queda resumido en la Figura 3.3.

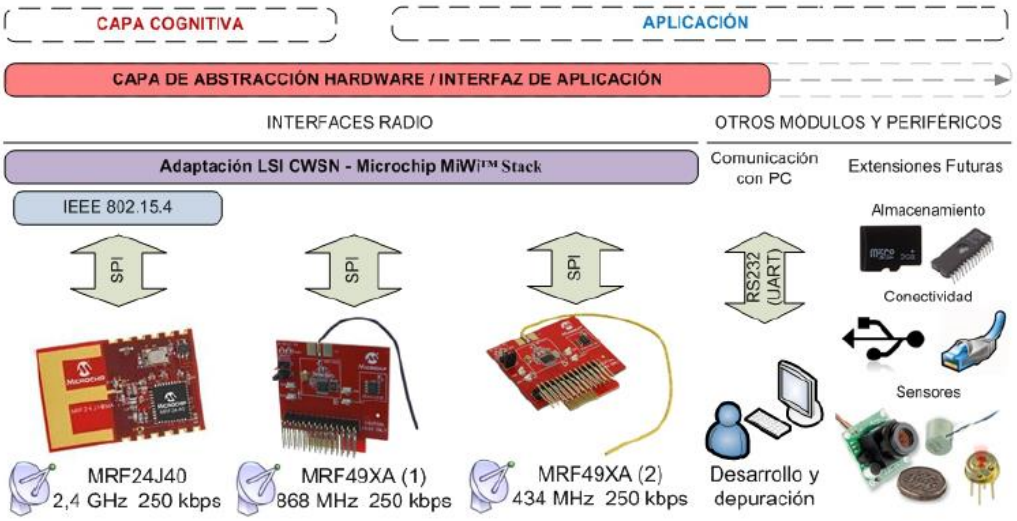
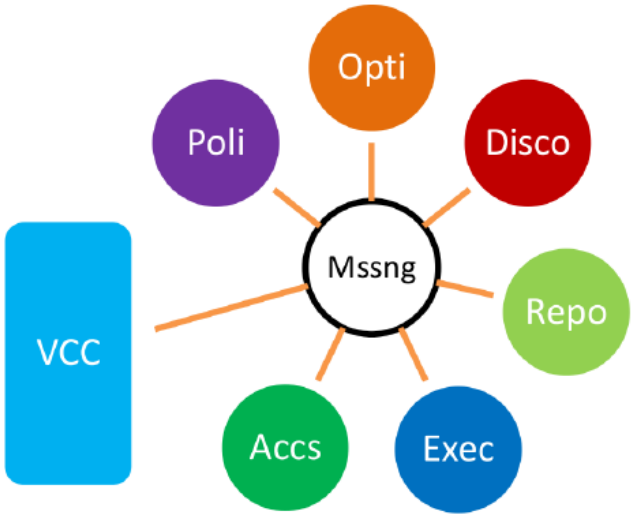


Figura 3.3

Las funciones implementadas actualmente en la HAL van desde la inicialización del nodo hasta la gestión de las comunicaciones (saber el canal activo en una interfaz, enviar y recibir paquetes, comprobar la tabla de conexiones, etc.).

3.3. Arquitectura cognitiva

La implementación de la arquitectura cognitiva realizada en [4] es la encargada de dar soporte a la implementación de estrategias cognitivas en el nodo. El esquema general de la arquitectura se puede ver en la Figura 3.4.



La función de cada sub-módulo de la arquitectura es:

- Repository. Sub-módulo encargado de almacenar la información necesaria para la estrategia cognitiva. Aquí se recibirá la información y se almacenará para cuando se la pida cualquier sub-módulo de la arquitectura.
- Discovery. Es el encargado de caracterizar diferentes parámetros del entorno. Este sub-módulo será el encargado de obtener el nivel de ruido en los canales.
- Optimization. En este sub-módulo se implementarán las rutinas de las estrategias cognitivas. Mientras este sub-módulo realiza el proceso cognitivo podrá realizar peticiones a otros sub-módulos del CRModule o incluso a otros sub-módulos en otros nodos.
- Execution. Los resultados del proceso de optimización tendrán que ser ejecutados. Este sub-módulo es el encargado de ejecutar las decisiones tomadas por el sub-módulo Optimization.
- Access Control. Debido a la naturaleza cooperativa de las estrategias cognitivas, son necesarios mecanismos de seguridad y control que sepan qué nodos tienen permisos para hacer acciones sobre el resto de nodos de la red. Éste sub-módulo se encarga de manejar la información de los permisos que tienen los nodos conocidos para realizar acciones en los sub-módulos del CRModule al que pertenece.
- Policy Support. Las estrategias cognitivas tienen unos valores que determinan las decisiones que se toman. Por tanto, este sub-módulo tiene la información sobre esos valores y es consultado por el resto de sub-módulos, sobre todo por Optimization, para tomar las decisiones oportunas.
- Messenger. Es el sub-módulo central de la arquitectura. Se encarga de conectar el resto de sub-módulos entre ellos. Maneja los mensajes que se mandan el resto de sub-módulos y comprueba, si el mensaje proviene de otro nodo, si tiene permisos o no mediante petición al sub-módulo Access Control.

La implementación realizada en [4] fue desarrollada para funcionar directamente sobre la pila de protocolos de Microchip y para funcionar sobre una plataforma hardware distinta. Por ello, posteriormente, en [5], se realizó una adaptación de esta arquitectura para funcionar con la pila de protocolos y el firmware mencionados en el apartado anterior.

3.4. Algoritmos a implementar

A continuación se resumirán los algoritmos que se han implementado en la realización de este trabajo, dejando los detalles y el resultado de las simulaciones para la consulta en los documentos referenciados.

3.4.1. Algoritmo de seguridad

Este algoritmo, desarrollado en [6], consiste en evitar que un nodo se haga pasar por usuario primario de la red, denegando el uso de la misma a otros nodos.

Para la detección de los nodos intrusos, o que tienen un funcionamiento anómalo, el algoritmo requiere de dos fases:

- Fase de aprendizaje. En esta fase el nodo procesa los paquetes que recibe, guardando el valor del RSSI y el tiempo que ha transcurrido entre dos paquetes. Con esto construye una lista de clústers que tienen de coordenadas el valor del RSSI y del tiempo entre paquetes y un radio. Para la creación de los clústers primero se normalizan los valores almacenados y luego se van procesando cada par de coordenadas incluyéndolas en un clúster.
- Fase de detección. Es el tiempo restante que el nodo esté trabajando. En esta fase el algoritmo se encargará de coger el valor del RSSI y tiempo entre paquetes de los paquetes que va recibiendo y comprobando que están contenidos en un clúster de los anteriores. Si un paquete no está contenido en un clúster, se marca el nodo del que procedía como atacante y se informa al resto de nodos de la red. Si un número determinado de nodos detectan a un mismo nodo como atacante desconectan ese nodo de la red.

3.4.2. Algoritmo de reducción de consumo

Este algoritmo, detallado en [7], trata de reducir el consumo de los nodos de la red mediante teoría de juegos.

El algoritmo aprovecha que es más costoso transmitir por un canal ruidoso que cambiar todos los nodos a un canal con mejor calidad de enlace. Por tanto, el algoritmo consiste en conocer el estado del canal por el que se está transmitiendo mediante el RSSI de los paquetes que se reciben, si se baja de un umbral predefinido, se lanzará el proceso de decisión de cambio de canal. La decisión se toma calculando los costes asociados a transmitir por un canal ruidoso, con un número de retransmisiones por paquete, y el coste de sensar el medio y enviar mensajes para cambiar de canal. Si se decide cambiar de canal, el coste de cambiar es menor que el coste de transmitir en el canal en el que se está transmitiendo, se procede a elegir el canal menos ruidoso y a decidir entre todos los nodos de la red a qué canal se cambia definitivamente.

Capítulo 4. Implementación del algoritmo de seguridad

En este capítulo se va a detallar la implementación del algoritmo descrito en [refPaperJavi]. Se van a detallar las funciones que se han añadido en la arquitectura cognitiva para lograr que los nodos detecten atacantes que se quieran hacer pasar por usuarios primarios de la red y los desconecten.

4.1. Funciones de la arquitectura cognitiva

Para la implementación de este algoritmo se ha decidido que se realicen los procesos necesarios en el sub-módulo Respository ya que es necesario el acceso a tablas de conexiones y pasos de direcciones que harían un código poco legible si se quisiera hacer en el sub-módulo Optimizer. Por esto, se ha decidido que Optimizer sea donde se coordinan las tareas del algoritmo pero que el procesamiento de los datos se haga en Repository.

Durante la ejecución del algoritmo se va a necesitar medir el tiempo durante el que se ha ejecutado el algoritmo y el tiempo entre paquetes de aplicación. Para medir el tiempo de ejecución del algoritmo se ha usado la rutina de atención a la interrupción del *timer 4*, que se ha configurado cada un milisegundo, ya que no se necesita una precisión muy elevada al ser necesario medir segundos.

Para medir el tiempo entre paquetes se ha decidido usar la función `MiWi_TickGet` de la librería `SymbolTime.c` que tiene una precisión de 16 microsegundos. Con esto podemos medir con precisión suficiente el tiempo transcurrido entre paquetes de aplicación que puede ir desde los pocos milisegundos hasta varios segundos.

Las etapas que sigue el algoritmo quedan resumidas en la Figura 4.1.

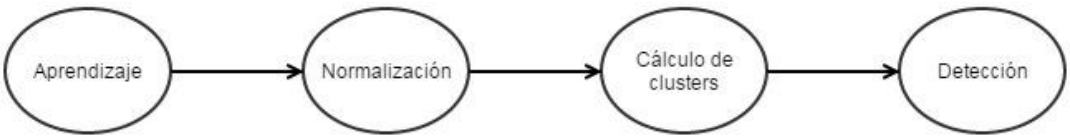


Figura 4.1 Etapas del algoritmo de seguridad

En cada paso de una etapa a otra se activa un *flag* que hace que no se vuelva a entrar a esa etapa ya que no es necesario volver nunca a una etapa anterior.

Las funciones implementadas en cada sub-módulo de la arquitectura cognitiva y los pasos de mensajes entre sub-módulos se exponen a continuación.

4.1.1. Optimizer

En este sub-módulo se va a realizar la coordinación de las etapas del algoritmo. Es el encargado de, cada vez que se reciba un mensaje de aplicación, enviar un mensaje al sub-módulo Repository para que lo procese. También se encarga de temporizar la ejecución de la etapa de aprendizaje, cambiando el *flag* que indica que se ha terminado la etapa. Este sub-módulo también envía mensajes a Repository para que se realice la etapa de normalización y de cálculo de clusters.

El tiempo que transcurre en la etapa de aprendizaje es ajustable mediante la variable `AprendizajeMax` definida en el archivo `ConfigOptimizer.h`. Una unidad de esta variable equivale a 50 milisegundos.

Otra tarea que realiza este sub-módulo es la de reiniciar la tabla donde se guardan las detecciones de atacantes que se han producido. Esto se puede modificar mediante la variable `ReinicioMax` en el archivo de cabecera anterior. Esto se ha hecho para que cada cierto tiempo se tenga que volver a detectar a un atacante, asegurándose así que las detecciones son correctas y evitando que un nodo que haya sido detectado incorrectamente como atacante no tenga la posibilidad de volver a conectarse a la red.

En la Figura 4.2 se resume el paso de mensajes entre los sub-módulos Optimizer y Repository con el campo DataType necesario para cada acción y la etapa en la que se encuentra el algoritmo cuando se envía esa petición.

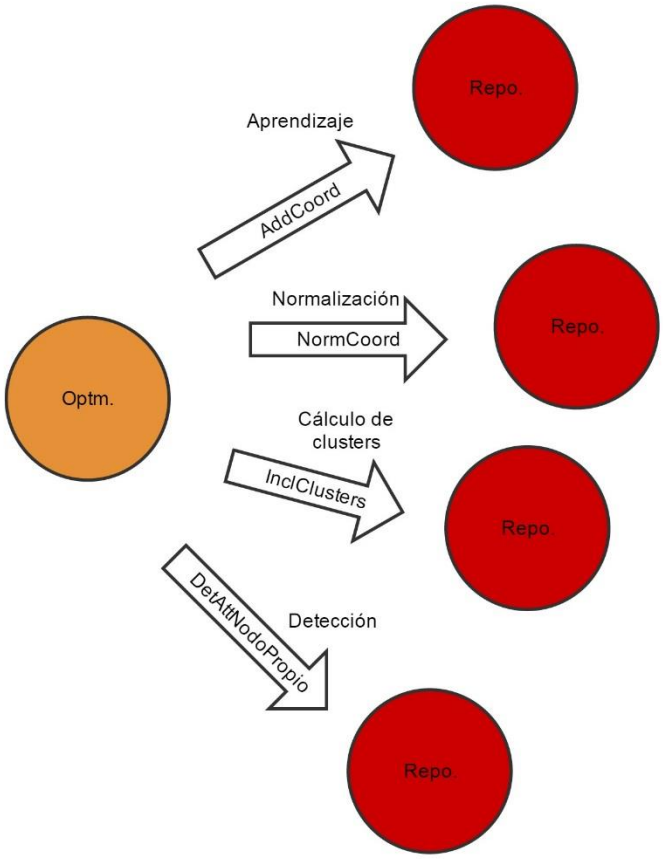


Figura 4.2 Diagrama del paso de mensajes entre Optimizer y Repository

4.1.2. Repository

A continuación se van a describir las estructuras que se han definido para almacenar los datos necesarios para la implementación del algoritmo y las funciones que se usan durante la ejecución del mismo.

Se han definido tres nuevas estructuras de datos:

- Coordenadas. En esta estructura se almacena la potencia de cada paquete recibido y el tiempo transcurrido entre éste y el anterior. Ambos parámetros se almacenan en una variable tipo *double* ya que serán necesarios decimales cuando se normalice. La etiqueta definida para esta estructura es "coord".

```
typedef struct coordenadas {
      double RSSI;
      double tiempo;
} coord;
```

- Clusters. En esta estructura se guardan los clusters generados durante la etapa de cálculo de clusters. Cada cluster está definido por un centro que es una coordenada definida anteriormente, un radio, de tipo double ya que es un número con decimales, y el número de paquetes con el que se ha formado el cluster necesario para hacer los cálculos que se detallarán en la explicación del método correspondiente. La etiqueta que define un cluster es "cl".

```
typedef struct cluster {
      coord centro;
      double radio;
      int nMuestras;
} cl;
```

- Atacantes. En las variables de este tipo se guardará la información relativa a la detección de un nodo atacante. Los parámetros almacenados son la dirección del nodo que se ha detectado como atacante, la dirección del nodo que lo ha detectado y un byte que indica si ha sido detectado como atacante o no. Éste último byte será el que se compruebe para saber cuántas veces se ha detectado el mismo nodo como atacante. La etiqueta para definir variables de este tipo es "at".

```
typedef struct atacantes {
      BYTE direccionAtacante[MY_ADDRESS_LENGTH];
      BYTE direccionDetector[MY_ADDRESS_LENGTH];
      BYTE esAtacante;
} at;
```

Para almacenar todos los datos recibidos se han definido tres listas:

- `coord Lista_Paq_Rec_Aprendizaje[MAX_PAQ_APRENDIZAJE]`
  Aquí se almacenará la información de cada paquete que se reciba durante la etapa de aprendizaje. El tamaño es fijo y definido por `MAX_PAQ_APRENDIZAJE` que se puede configurar en el archivo `ConfigRepository.h`.
- `cl Lista_Clusters[MAX_CLUSTERS]`
  En esta lista se guardarán los clusters que se generen a partir de los paquetes de la lista anterior. En este caso también tendrá un tamaño fijo y definido por `MAX_CLUSTERS` que también se puede configurar en el archivo `ConfigRepository.h`.

- `at Tabla_Atacantes[(CONNECTION_SIZE+1)*(CONNECTION_SIZE+1)]`
  Esta lista guardará los datos de los atacantes detectados por el resto de nodos de la red y por el propio nodo que almacena la lista. En este caso el tamaño dependerá del número de nodos que tenga en la tabla de conexiones y variará cada vez que se conecte un nodo. Inicialmente la tabla contiene las direcciones de todos los nodos tanto en el campo de `direccionAtacante` como en el de `direccionDetector` y el campo `esAtacante` a cero.

Una vez definidas todas las listas a utilizar en el algoritmo se pasa a detallar cada una de las funciones implementadas:

- `void inicializarTablaAtacantes()`
  Es la función encargada de inicializar la tabla de atacantes al iniciar el nodo y cuando se pasa el tiempo prestablecido para reiniciarla. La inicialización se hace de la siguiente manera:
  1) Se crea una tabla con las direcciones de todos los nodos de la red, incluyendo la dirección del propio nodo.
  2) Se rellena el campo direccionAtacante de todas las entradas de la tabla de atacantes con todas las direcciones de la tabla anterior repetidas el número de conexiones que haya más uno, ya que el propio nodo también cuenta, y se inicializa el campo esAtacante a cero.
  3) Se vuelve a recorrer la tabla esta vez rellenando el campo direccionDetector con una dirección de la tabla creada en el punto 1) hasta introducirlas todas y repitiéndolas hasta llegar al final.

  Un ejemplo en el que un nodo sólo estuviese conectado a otro la tabla de conexiones quedaría de la siguiente forma:

		direccionDetector	
		Dir. nodo 1	Dir. nodo 2
direccionAtacante	Dir. nodo 1	0	0
	Dir. nodo 2	0	0

Tabla 4.1 Ejemplo de tabla de atacantes inicializada con dos nodos en la red

- `BOOL CRM_Repo_Calculo_Coordenadas()`
  Esta función se ejecuta cada vez que se recibe un paquete de aplicación cuando se está en la etapa de aprendizaje. Se encarga de obtener la potencia y el tiempo transcurrido entre dos paquetes y almacenarlo en la lista de paquetes recibidos. Con cada potencia y tiempo obtenidos, se comprueba si es mayor que el máximo de entre los que se han procesado y si lo es lo guarda para después normalizar.
- `void CRM_Repo_Normalizar_Coordenadas()`
  Esta es la función que se ejecuta cuando el algoritmo se encuentra en la etapa de normalización. Para normalizar las coordenadas obtenidas se recorre la lista de paquetes recibidos dividiendo cada valor de potencia y tiempo entre el máximo obtenido durante la etapa de aprendizaje. Los valores resultantes se vuelven a almacenar en la lista de paquetes recibidos.
- `void CRM_Repo_Calculo_Clusters()`
  Esta función es la encargada de crear los clusters que se usarán posteriormente para detectar si un nodo es atacante o no. Para ello, se recorre la lista de paquetes recibidos comprobando si pertenece a un cluster ya creado o no. Para saber si un paquete

pertenece a un cluster se calcula la distancia entre el centro de cada uno de ellos y el paquete y si es menor que el radio se dice que pertenece al cluster. Si un paquete no pertenece a ninguno, se crea uno nuevo con centro las coordenadas de ese paquete y un radio inicial configurable. Cuando un paquete pertenece a alguno ya creado, se añade al cluster calculando el nuevo centro y radio usando las siguientes fórmulas:

$$radio = radio\ anterior + distancia$$

$$RSSI_{nuevo} = \frac{(RSSI_{cluster} * n) + RSSI_{paquete}}{n+1}$$

$$tiempo_{nuevo} = \frac{(tiempo_{cluster} * n) + tiempo_{paquete}}{n+1}$$

Ecuación 4.1 Cálculo de nuevo radio y centro cuando se recibe un paquete que pertenece a un cluster

Cuando termina de procesar todos los paquetes que se han recibido quedaría un mapa de clusters como el de la Figura 4.3.

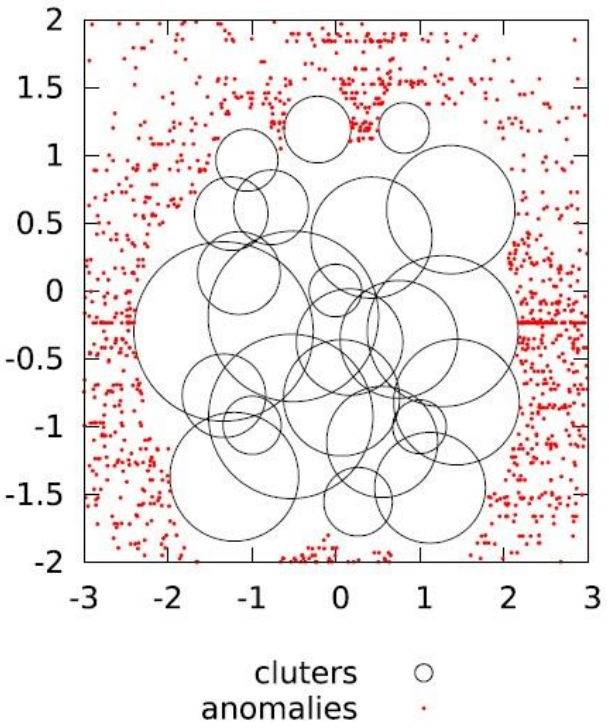


Figura 4.3 Ejemplo de mapa de clusters, obtenido de [refPaperJavi]

- `double CRM_Repo_Calculo_Distancia(coord pto1, coord pto2)`
  Esta es una función auxiliar para calcular la distancia de dos puntos con dos dimensiones, en nuestro caso las coordenadas de los paquetes y de los centros de los clusters. Se llama a esta función cada vez que se tiene que comprobar que un paquete recibido pertenece a un cluster. La ecuación para calcular la distancia es la siguiente:

$$distancia = \sqrt{(RSSI_{cluster} - RSSI_{paquete})^2 + (tiempo_{cluster} - tiempo_{paquete})^2}$$

Ecuación 4.2 Cálculo de la distancia entre dos puntos

- `BOOL CRM_Repo_Detectar_Atacante()`
  Esta función es la encargada de procesar los mensajes de aplicación que llegan durante la fase de detección del algoritmo. Con cada paquete comprueba que pertenezca a un cluster y si no pertenece a ninguno añade al nodo que lo envió a la lista de atacantes. Tras añadirlo a la lista se manda un mensaje al resto de nodos notificando que se ha detectado un nodo como atacante.
- `BOOL CRM_Repo_Proc_Mens_Att(REPO_MSSG_RCVD *Peticion)`
  Función encargada de procesar los mensajes con datos de atacantes procedentes de otros nodos. La finalidad es la misma que en el caso anterior. Cuando se ha incluido el nodo atacante en la tabla correspondiente se manda la información a los nodos que estén conectados al que recibe el mensaje difundiéndose así el mensaje por toda la red. Al hacer esto se pueden recibir varios mensajes iguales por lo que se comprueba si el campo esAtacante es cero antes de repetir el mensaje al resto de nodos.

Capítulo 5. Implementación del algoritmo de reducción de consumo

En este capítulo se va a detallar el trabajo desarrollado para la implementación del algoritmo desarrollado en [refPaperElena]. Se explicarán las nuevas funciones añadidas en el CRModule y los mensajes y respuestas que se intercambian los nodos para conseguir el cambio a un canal más óptimo para la transmisión de mensajes de aplicación.

5.1. Funciones de la arquitectura cognitiva

La implementación de este algoritmo se ha realizado siguiendo un esquema de máquinas de estados finitos tal y como se presenta en la Figura 5.1.

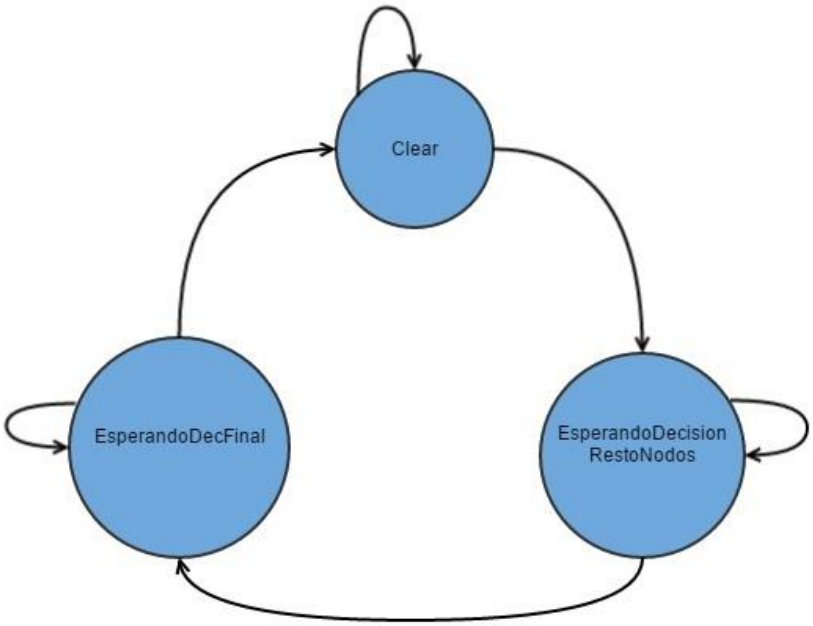


Figura 5.1 Diagrama de estados de la implementación propuesta

Las transiciones entre estados de la Figura 5.1 dependen de las respuestas de los otros nodos a las peticiones de cambio de canal que se les hace. Todos los nodos comenzarán en estado "Clear" en el que comprobarán, con cada mensaje de aplicación que envíen, el número de retransmisiones que se han realizado. Tras enviar el mensaje de aplicación y en la ejecución de la rutina de atención a la interrupción del *timer* 4, que se produce cada milisegundo, el nodo calcula los costes asociados a la transmisión en el canal en el que está transmitiendo y los costes asociados al cambio de canal. Cuando el coste de cambio es menor que el coste de transmitir en el canal actual, ese nodo inicia el proceso de elección del canal al que cambiar y notifica al resto de nodos en su red.

El paso de mensajes de petición de cambio de canal y las respuestas se producen a través de VCC, habiendo reservado la interfaz de 434 MHz disponible en los nodos para tal efecto.

A continuación se pasa a describir las funciones de cada uno de los sub-módulos del CRModule para procesar las peticiones de cambio y obtener el resultado final.

5.1.1. Optimizer

El sub-módulo Optimizer es el encargado de la ejecución del algoritmo. Decidirá el inicio del cambio de canal, procesará las respuestas de los otros nodos de la red y pedirá al resto de sub-módulos la ejecución de determinadas acciones o la información que necesite durante el proceso. La estructura de los mensajes dirigidos a este sub-módulo y que se usará posteriormente para enviarle información desde otros sub-módulos es la siguiente:

```
typedef struct _OPTM_MSSG_RCVD
{
      INPUT BYTE OrgModule;
      INPUT OPTACTION Action;
      IOPUT void *Param1;
      IOPUT void *Param2;
      INPUT radioInterface Transceiver;
      INPUT BYTE *EUINodo;
} OPTM_MSSG_RCVD;
```

Las funciones que se han implementado o modificado en este sub-módulo se exponen a continuación:

- `BOOL CRM_Optm_Cons(OPTM_MSSG_RCVD *Peticion)`
  Es la función encargada de pedir al sub-módulo Discovery la información del ruido en los canales de las interfaces de 868 MHz y 2,4 GHz, obteniéndose así un canal óptimo por cada interfaz y el valor del RSSI del ruido en ese canal. De esos dos canales se elegirá el más óptimo y se guardará la interfaz a la que pertenece.
  Una vez hecho esto se distingue entre si el algoritmo ha sido iniciado en el propio nodo, es decir, el parámetro Action del mensaje que se ha recibido era SubActCambio, y lo que hace es enviar un mensaje por VCC dirigido al sub-módulo Repository del resto de nodos con la información sensada. O por el contrario, lo que ha iniciado el algoritmo ha sido un mensaje de otro nodo, que lo que hace es procesar la información que se ha recibido, detectando si el canal que ha recibido del otro nodo estaba entre sus cuatro mejores (o con un valor de ruido muy próximo al de su cuarto mejor) y acepta el canal propuesto o le manda su mejor canal al resto de nodos. Esto último se realiza cuando el parámetro Action del mensaje que recibe es SubActRespCambio. Todo esto queda resumido en la Figura 5.2.

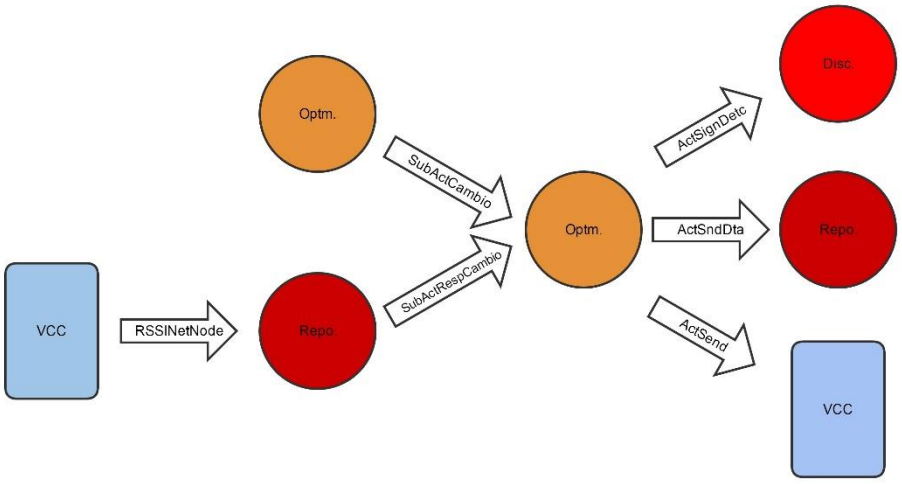


Figura 5.2 Diagrama con las acciones que realiza la función Cons y los parámetros Action que se pasan.

- `BOOL CRM_Optm_Calcular_Costes(BYTE n_rtx)`
  Esta función se encarga de calcular los costes asociados a la transmisión en un canal ruidoso y los costes de cambiar de canal para decidir si iniciar o no el proceso para cambiar de canal. Se pide de argumento el número de retransmisiones del último mensaje de aplicación que se ha intentado enviar y el valor de salida es *TRUE* o *FALSE* dependiendo de la decisión de cambio de canal.

- `BOOL CRM_Optm_Processor(OPTM_MSSG_RCVD *Peticion)`
  Esta función es la que se encarga de procesar los mensajes provenientes de otros nodos. Existen tres tipos de mensajes de respuesta, una petición de cambio de canal, una respuesta a la petición de cambio o una decisión final.
  Dependiendo del estado en el que se encuentre el nodo receptor y del mensaje que reciba se realizará una acción u otra. Las acciones realizadas se detallan a continuación:
  - Estado “Clear”. Cuando se esté en este estado sólo se procesarán los mensajes de petición de cambio de canal. El parámetro Action del mensaje dirigido al sub-módulo Optimizer tiene que ser ProcCambioCanal.
    Cuando llegue un mensaje con ese parámetro se calculan los costes de cambio de canal y de transmisión en el canal actual y se manda respuesta al resto de nodos. Si la respuesta es positiva se llama a la función `CRM_Optm_Cons` para que se realice el sensado del medio se decida el canal óptimo. Todo esto queda resumido en la Figura 5.3.

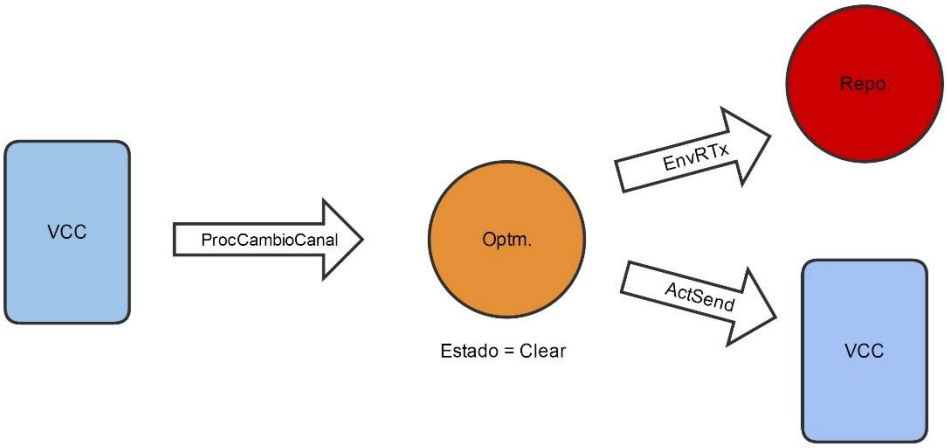


Figura 5.3 Diagrama de mensajes entre sub-módulos cuando se recibe una petición de cambio de canal

  - Estado “EsperandoDecisionRestoNodos”. En este estado se esperan las respuestas al cambio de canal de todos los nodos que estén conectados. Si se recibe una respuesta negativa se evalúa el porcentaje de mensajes de aplicación que se han intercambiado los dos y si hay uno con el que se comunica mucho se rechaza el cambio de canal; si no se comunica mucho sigue con el proceso de cambio. El parámetro Action del mensaje dirigido al sub-módulo Optimizer tiene que ser ProcCambioCanal.
    Los pasos de mensajes entre los sub-módulos cuando se recibe una respuesta a un mensaje de cambio de canal queda resumido en la Figura 5.4.

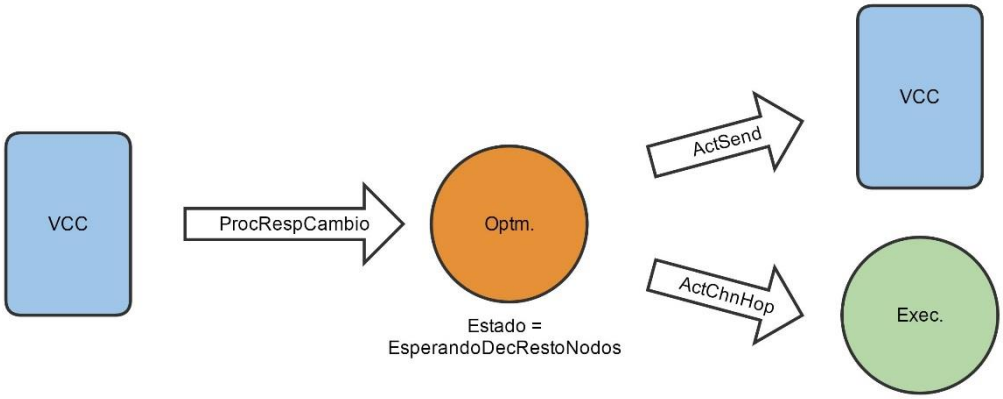


Figura 5.4 Diagrama de mensajes entre sub-módulos cuando se recibe una respuesta a una petición de cambio de canal

- Estado "EsperandoDecFinal". En este estado todos los nodos en la red deciden a qué canal cambiarse. Cada nodo recibe la información del espectro del resto de nodos y comprueba si el canal elegido está entre sus cuatro mejores. Si está entre los cuatro manda un mensaje al resto de nodos confirmando que se va a cambiar y se cambia de canal. Si no está entre esos cuatro comprueba si el nivel de ruido del cuarto mejor canal es parecido al del canal que le han enviado y si lo es acepta el cambio. El resultado de este estado siempre va a ser un cambio de canal. El parámetro Action del mensaje dirigido al sub-módulo Optimizer tiene que ser ProcRespCambio si es un mensaje de respuesta afirmativa o negativa a una petición, o ProcDecFinal si ya se ha recibido el canal óptimo de ese nodo y se ha decidido no cambiar a ese. Los pasos de mensajes entre sub-módulos cuando se recibe una respuesta con un canal distinto al que se propuso inicialmente viene resumido en la Figura 5.5.

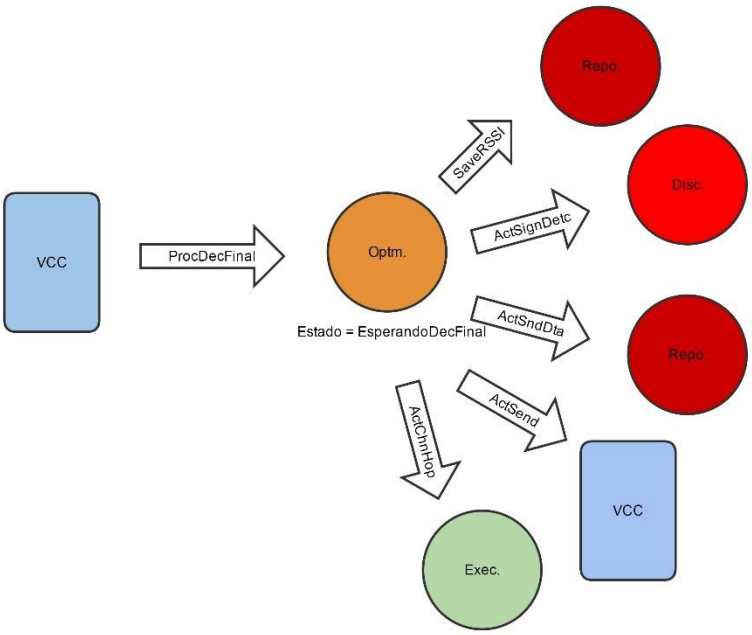


Figura 5.5 Diagrama de mensajes entre sub-módulos cuando se recibe una respuesta de cambio de canal con un canal diferente al propuesto inicialmente

5.1.2. Repository

Este sub-módulo será el encargado de almacenar la información que se reciba del resto de nodos de la red y enviarla al resto de sub-módulos si se la piden. La estructura de los mensajes con destino este sub-módulo es la siguiente:

```
typedef struct _REPO_MSSG_RCVD
{
      INPUT BYTE OrgModule;
      REPACTION Action;
      INPUT BYTE *EUINodo;
      INPUT REPODATATYPE DataType;
      INPUT radioInterface Transceiver;
      IOPUT void *Param1;
      IOPUT void *Param2;
      IOPUT void *Param3;
      IOPUT void *Param4;
} REPO_MSSG_RCVD;
```

Cuando le lleguen mensajes de VCC los datos se almacenarán en Param2 como se explicará en la siguiente sección y cuando le lleguen mensajes de Optimizer devolverá la información en diferentes parámetros dependiendo del dato pedido. La información que se puede pedir y almacenar en Repository y las funciones implementadas se detallan a continuación:

- `BOOL CRM_Repo_SendDat(REPO_MSSG_RCVD *Peticion)`
  Esta función es la encargada de enviar la información que se le pide al sub-módulo Repository desde otros sub-módulos. Para la implementación de este algoritmo se ha añadido la posibilidad de devolver el número de retransmisiones en un canal, el canal y el RSSI en una posición de la lista de canales ordenada y el número de mensajes intercambiados con otro nodo.
  Para devolver el canal y el RSSI de una posición, se ordena la lista de RSSI de menor a mayor, ocupando la menor potencia la posición cero, y ordenando una lista con los números de canal de la misma forma que la de RSSI. El algoritmo de ordenación usado para esto ha sido *Bubble Sort.*
- `void CRM_Repo_NodoPropio(REPO_MSSG_RCVD *Peticion)`
  En esta función se ha añadido la opción de almacenar el número de retransmisiones de un paquete de aplicación mediante el envío de un mensaje a Repository con el campo Param1 del mensaje con el valor EnvRTx.
- `BOOL CRM_Repo_NodosRed(REPO_MSSG_RCVD *Peticion)`
  Esta función se encarga de recibir los mensajes de otros nodos con información de sensado del otro nodo. Almacena el valor del canal óptimo y de su potencia de ruido para las interfaces de 868 MHz y 2,4 GHz además del mejor canal de entre esos dos y la interfaz a la que pertenece. Dependiendo del estado en el que se encuentre el nodo en el momento en que recibe el mensaje, manda una petición al método `CRM_Optm_Processor` con diferente Param1 para que el sub-módulo Optimizer procese la información del mensaje.
  El paso de mensajes entre Repository y Optimizer cuando recibe información del espectro de otros nodos y el campo Param1 para cada estado queda detallado en la Figura 5.6.

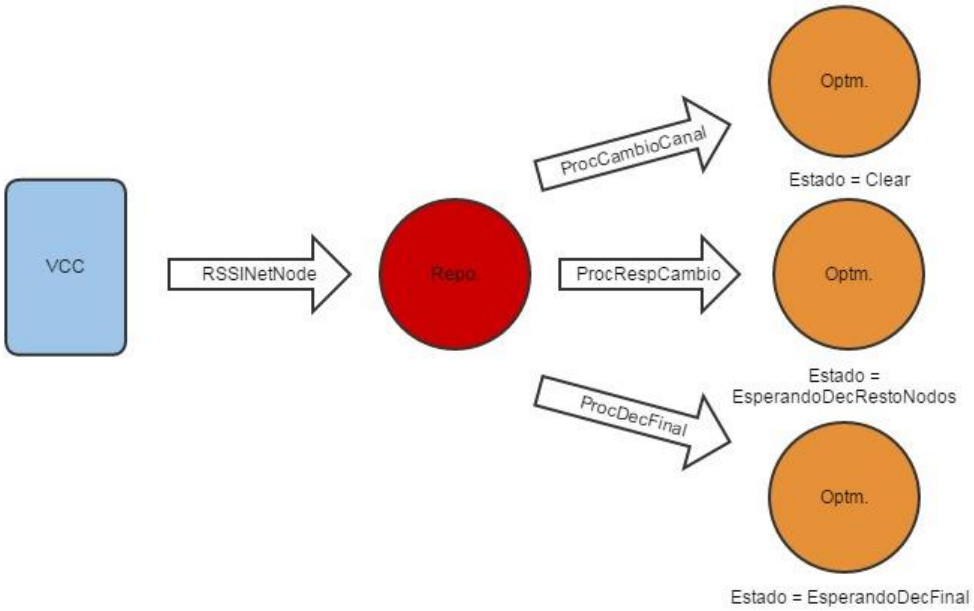


Figura 5.6 Diagrama de peticiones a Optimizer cuando se recibe información de sensado de otro nodo

- `void CRM_Repo_NRTx(BYTE n_rtx, BYTE canal, radioInterface ri)`
  Es la función encargada de almacenar el número de retransmisiones que se producen al enviar un mensaje de aplicación. Los parámetros de entrada son la interfaz radio con la que se está transmitiendo, el canal en el que se han producido las retransmisiones y el número total de retransmisiones que se han producido.

- `void CRM_Repo_Mensajes_Intercambiados(BYTE *Address)`
  Esta función es la encargada de actualizar el número de mensajes que se han recibido de un mismo nodo durante la ejecución de la aplicación. La llamada a esta función se realiza mediante un mensaje a Repository con el campo DataType con el valor AddMsg. El mensaje se envía cuando en la interrupción del timer 4 se comprueba si hay datos en el buffer de recepción.
- `void CRM_Repo_Str_RSSI(radioInterface ri)`
  Esta función se encarga de guardar el valor de la potencia de ruido térmico obtenida del sensado del espectro realizado anteriormente por el sub-módulo Discovery. Posteriormente se utiliza esta información para decidir si el canal al que quiere cambiar otro nodo de la red es adecuado para el nodo que hace el sensado.
- `BOOL CRM_Repo_Reiniciar_RTx(void)`
  Es una función auxiliar. Se llama a esta función cada vez que se decide no cambiar de canal debido a que se decide no cambiar de canal porque un nodo con el que se ha intercambiado un porcentaje elevado de mensajes ha decidido no cambiar de canal. Poniendo a cero el número de retransmisiones producidas se evita que se inicie el algoritmo dos veces consecutivas sin que se intente enviar un mensaje de aplicación nuevo.

5.1.3. VCC

En este sub-módulo se ha tenido que modificar el modo en el que se pasaban los mensajes recibidos con destino Repository. Al tener que enviar un número de datos superior al número de campos en el mensaje con posibilidad de incluir datos personalizados, y al disponer de

punteros con la capacidad de incluir cualquier tipo de dato, se ha procesado la información útil de cada mensaje, incluyéndolo en un vector y pasando ese vector a través del puntero al sub-módulo destino.

5.1.4. Execution

La funcionalidad de este sub-módulo no ha sido modificada. Las peticiones que se le hacen desde Optimizer son para cambiar el canal en el que se está transmitiendo.

5.1.5. Discovery

La funcionalidad de este sub-módulo tampoco ha sido modificada. Los mensajes que le envía Optimizer son para pedir la información del canal óptimo de cada interfaz y el valor de ruido térmico que reciben.

5.2. Comentarios

La funcionalidad de enviar mensajes *broadcast* mediante el sub-módulo VCC no estaba implementada en la HAL ni en el sub-módulo por lo que se ha modificado la estructura de los mensajes que se envían a VCC añadiendo el parámetro AddrMode. Además se ha modificado la función `Send_Buffer` en la HAL añadiendo un parámetro de entrada con el mismo nombre. Con esto, pasando el parámetro BROADCAST_ADDRMODE a los mensajes dirigidos a VCC se consigue la nueva funcionalidad.

Capítulo 6. Aplicación de prueba de los algoritmos

6.1. Diseño

6.2. Implementación

Capítulo 7. Conclusiones y líneas futuras

7.1. Conclusiones

7.2. Líneas futuras

Bibliografía

Lista de acrónimos