# 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.

## 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.

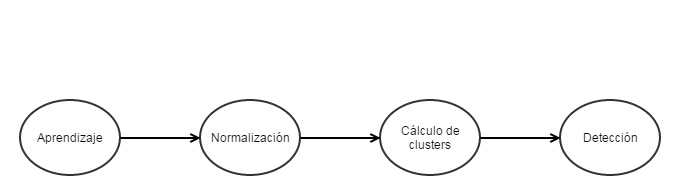


Figura 4.1 Etapas del algoritmo de seguridad

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.

### 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.

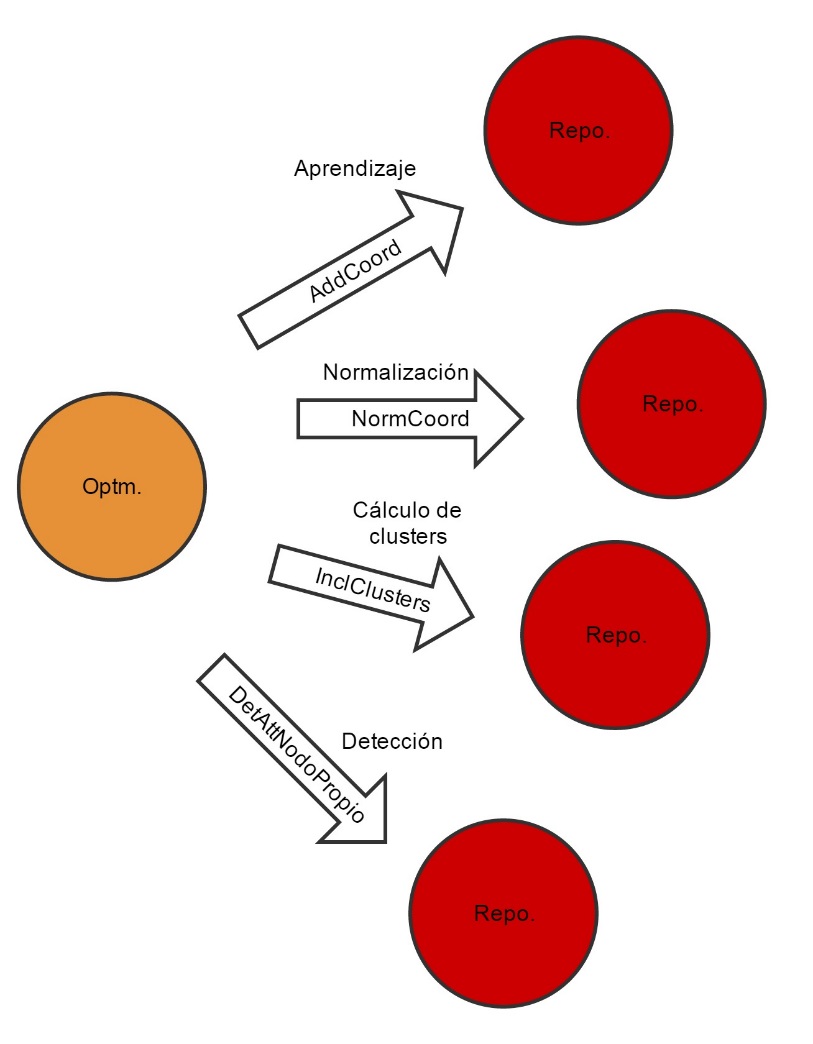


Figura 4.2 Diagrama del paso de mensajes entre Optimizer y Repository

### 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:

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.

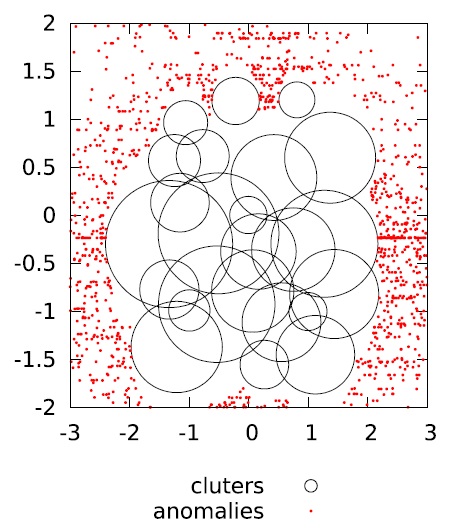


Figura 4.3 Ejemplo de mapa de clusters, obtenido de [refPaperJavi]

* 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:

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.