Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros de Telecomunicación

Proyecto fin de Carrera

Ingeniero de Telecomunicación

Implementación de una arquitectura para el desarrollo de capacidades cognitivas en teléfonos inteligentes

César Villar García

Agosto 2014

Proyecto fin de Carrera

Título: Implementación de una arquitectura para el desarrollo de capacidades cognitivas en teléfonos inteligentes

Autor: César Villar García

Tutor: Javier Blesa Martínez

Departamento: Departamento de Ingeniería Eletrónica

Tribunal Proyecto fin de carrera

Presidente: D. Octavio Nieto-Taladriz García

Miembro: D. José Manuel Moya Férnandez

Secretario: D. Alvaro Araujo Pinto

Suplente: D. Pedro José Malagón Marzo

PUNTUACIÓN:

Siguenza, a 17 de agosto de 2014

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros de Telecomunicación

Proyecto fin de Carrera

Ingeniero de Telecomunicación

Implementación de una arquitectura para el desarrollo de capacidades cognitivas en teléfonos inteligentes

César Villar García

Agosto 2014

Resumen

PALABRAS CLAVE: *redes cognitivas, radio cognitiva, redes de sensores inalámbricas, redes de sensores inalámbricas cognitivas. Android*

To My Son...

Agradecimientos

Lo esencial nunca está en la partitura.
George Steiner

A todas las personas que forman parte del Laboratorio de Aplicaciones Bioacústicas por su apoyo, información, orientación, asesoramiento y acompañamiento a lo largo de este proyecto, confiando plenamente en mí y en mi toma de decisiones. A Michel por darme esta oportunidad y a Joan por su tutorización, ayuda, y por todo el tiempo y consejos que me ha dedicado. Gracias también a: María, Marta, Mike, Serge, Alex y Ludwig, que habéis hecho del laboratorio un segundo hogar. Sin duda me habéis impresionado; no solo como grupo de investigación, también, y sobre todo, por vuestro valor humano. Recordaré este año en el LAB con mucho cariño.

Índice

Índice de Figuras

Índice de Tablas

Capítulo 1

Diseño

No es pretender el triunfo,
es resistir a los fracasos

Def. Esperanza, Novena Virgen de la Salud

En este capítulo definimos la arquitectura del nodo cognitivo sobre la que detallamos las interfaces de programación y los procesos de red invulcrados en su funcionamiento.

1.1. Arquitectura cognitiva

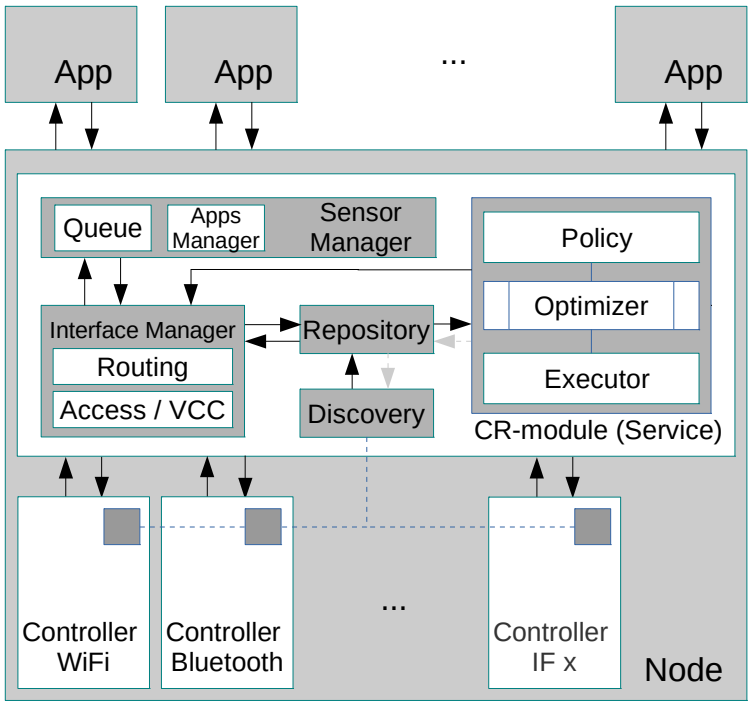

Figura 1.1: Arquitectura cognitiva

Nuestra arquitectura se ha diseñado siguiendo pinceladas del modelo CB (*Connectivity Brokerage*) [1], no se han podido seguir todas las recomendaciones puesto que no implementamos el objeto `CompNet` (distintas redes cooperando para por ejemplo mitigar la interferencia) y que no tenemos acceso a otros planos que no sea el de aplicación. Al tener *hardware* sin posibilidad de cambios y al tener un control limitado de éste (sólo las partes que *Android* nos permite) se desvanece la posibilidad de selecionar proactivamente la modulación, la codificación o el nivel de potencia en la transmisión. Nos restringimos a el diseño de una arquitectura confiando en las propias capacidades cognitivas u optimizaciones que elige la plataforma de por sí o que son accesibles de alguna u otra manera: *reflection*, *root*...

Como puede verse en la figura 1.1 se ha estructurado la arquitectura en dos capas, existiendo una subdivisión de la capa inferior en otras dos. Un modelo de capas permite definir interfaces de comunicación entre las diferentes unidades, limitando el alcance y conocmiento de cada módulo independizando su propósito del resto. A saber:

1. '**Capa de aplicación**': Formada por cada una de las aplicaciones que operan sobre el servicio cognitivo que les sirve como canal de comuncaciones. Cada una de estas aplicaciones, tiene que cumplir con la interfaz definida entre esta capa y la inferior (aplicación y cognitiva). El diseño e implementación de cada aplicación excede a este proyecto y depende en cada caso del tercero que desarolle la aplicación. No obstante para probar y utilizar las capacidades de nuestro nodo cognitivo se ha desarrollado una aplicación cuyo diseño presentamos en la sección 1.4 e implentación en 2.3 en el siguiente capítulo.

2. '**Capa cognitiva**' El interés del proyecto se centra en el desarollo de esta capa. Es una capa compuesta por otras dos: una primera capa de control donde se recoge toda la inteligencia y propósitos cognitivos y otra capa de actuación que provee de toda la funcionalidad para recibir y enviar datos por cada una de las interfaces de comunicación presentes:

    I. '**Módulo cognitivo**' Está fuertemente inspirado en los módulos que podemos ver en la arquitectura cognitiva desarrollada sobre Castalia [2] con ciertas peculiaridades e implementaciones que veremos a continuación.

    II. '**Controladores de interfaz**' Esta capa está formada por cada una de las interfaces presentes en *smartphone*, la implementación final recoge la interfaz de *WiFi* y *Bluetooth*, pero el diseño recoge un número arbitrario debido a que si construimos este módulo siguiendo la interfaz entre capas, puede ser fácilmente añadida a la arquitectura final.

A continuación haremos una breve descripción de los diferentes bloques que hemos definido en la arquitectura, al ser los procesos entidades que hacen uso más de un bloque se ha preferido dejar para más adelante (sección 1.3) su explicación una vez vista la arquitectura. Así se obtienen varios puntos de vista que ayudan a comprender el funcionamiento.

1.1.1. Interfaz Aplicación - Módulo Cognitivo

El apellido cognitivo supone una abstracción en el modelo de comunicaciones de tal forma que el usuario (la aplicación) sea transparente a los parámetros, tiempos

o detalles de la comunicación usada para transmitir los datos. Para lograr esta abstracción necesitamos un punto de ruptura que aisle y defina solamente la información estrictamente necesaria a intercambiar entre ambos agentes para poder recrear una CWSN (*Cognitive Wireless Sensor Network*). En la sección 1.2 se expone la definición de la API (*Application Programming Interface*) que debe cumplir el desarrollador de las distintas aplicaciones.

Al definir una interfaz permitimos que terceros puedan desarollar sus propias aplicaciones que usen de este servicio para montar sus comunicaciones, podemos decir que es una interfaz tipo cliente-servidor, donde se permite dar soporte simultáneo a varias aplicaciones, esto es posible gracias al `Apps Manager` que explicamos en la siguiente sección.

1.1.2. Sensor Manager

Es el punto de entrada a nuestro servicio cognitivo, el encargado de recoger las peticiones de las distintas aplicaciones e informar tanto de la respuesta a la aplicación que lo solicitó, como al resto de ellas si las hubiese acerca de información común generada en la CWSN donde cooperan. Esto se consigue gracias al submódulo `Apps Manager` que lleva un registro de cada una de las aplicaciones y permite discernir entre peticiones de éstas. Las peticiones entrantes son sanetizadas y codificadas para poder disparar los procesos internos a lo largo de los diferentes módulos y descodificadas elimando cualquier rasto cognitivo antes de ser entregadas a la aplicación en cuestión. La presencia del submódulo `Queue` permite, entre otras funciones que se detallarán, encolar mensajes de aplicación por si estas peticiones no pudiesen ser atendidas en el mismo momento de la recepción (fundamentalmente debido a cambios de contexto o *handovers*. Una vez que las condiciones lo permitan se consulta a esta cola para ver si quedan ordenes pendientes.

1.1.3. Policy, optimizer, executor

Estos tres módulos podemos considerarlos como el cerebro cognitivo, su interacción es jerárquica. De acuerdo a unas políticas descritas en el módulo `policy`, el `optimizer` evalua el cumplimiento de ellas conforme a valores que recoge del módulo `repository` que sirve como punto concentrador de la información, esta evaluación puede generar nuevos valores a almacenar en el repositorio y/o la actuación directa gracias a el módulo `executor`.

Para nuestro primer diseño sólo hemos definido una política cognitiva (ver sección 1.3.7) y el módulo ejecutor sólo realiza cambios de cambio de contexto o *handovers*, debido a las limitaciones en la plataforma para ajustar otros parámetros que finalmente han sido descartados del diseño.

1.1.4. Repositorio

La información que hemos de guardar en el repositorio es:

Información relativa al servicio Necesitamos almacenar interfaz actual de comunicaciones y el estado de ésta, así como un valor que nos permita averiguar si el servicio ha sido destruido por causas ajenas para intentar reconstruir la CWSN

ante una eventualidad, esta información necesita ser persistente. También almacenamos cuál es el peridodo con la que el optimizador y ejecutor actúan en base al tiempo. Si en vez de sobreescribir esta información hacemos un seguimiento, podemos obtener información acerca de la estabilidad y la adaptabilidad de la red.

Información relativa a los nodos en la red Para hacer un mapa de red necesitamos información acerca de qué nodos hay en la red y cómo llegar a ellos, los datos elegidos son:

- Identificador del nodo. Un entero asignado al conocer el nodo por primera vez.
- Nombre del nodo. Una *cadena de caracteres* representando el nombre con el que se presenta a la red.
- Papel del nodo. Una *cadena de caracteres* cuya codificación es: 'p' para usuario primario y 's' como secundario ('u' para valor desconido).
- Tipo de nodo. Una *cadena de caracteres* cuyos valores pueden ser: 'n' normal, 'c' coordinador ('u' para valor desconocido).
- Dirección MAC (*Medium Access Control*) de *Bluetooth* del nodo. Este dato identifica al nodo unívocamente
- Dirección IP (*Internet Protocol*) del nodo.
- Identificador de su nodo coordinador. Un entero que hace referencia a un identificador de nodo previamente conocido y que nos permite alcanzar todos sus datos.

Información relativa al sensado Los datos recogidos por cada dispositivo los almacenamos siguiendo esta estructura:

- Identificador del nodo, un entero que nos indica quién es el propietario de la información.
- RSSI, un entero con el valor en dBm de la intensidad de señal recogida de la red *WiFi*.
- Tasa de envío, Un *double* con el intervalo de tiempo medio de envios en la red.
- Marca de tiempo, una referencia al momento en que estos datos fueron obtenidos.

Estos datos son compartidos con el coordinador quién distribuye (sin filtrar, aunque podría implementarse un filtrado ya que se han reservado valores *dummy* encaminados a el ocultamiento de la información) esta información. Nuestro repositorio por tanto es distribuido de tal forma que todos tengan la misma foto aunque no tengan los mismos valores, ya que los identificadores de los nodos en cada dispositivo varían, la clave que nos permite sincronizar todo el repositorio recae en la dirección MAC de *Bluetooth* que se usa como clave primaria.

1.1.5. Discovery

Este módulo se asemeja a los ojos de nuestra arquitectura, siguiendo con el anterior símil del cerebro, una ventana al mundo. Ambos módulos se conectan a través del

repositorio (una especie de bus). En la sección 1.3.7, se entra en detalle sobre los parámetros recogidos del entorno. Este módulo no sólo recoge parámetros, si no que monitoriza el estado de las diferentes interfaces de comunicaciones, por ello parte de este módulo está muy ligado al controlador de la interfaz, sin embargo debido a que ciertas políticas necesitan que el optimizador evalue parámetros de interfaces de comunicaciones no activas, se ha sacado del diseño del controlador esta parte para añadirla al modulo `discovery` y que pueda ser manejada en cualquier momento, en la sección 2.1.7 del siguiente capítulo completamos el por qué de ésto.

1.1.6. Interface Manager

Este módulo es el punto de salida del módulo o servicio cognitivo hacia los controladores. En confluyen dos planos: el primero para los mensajes de aplicación provenientes del `Sensor Manager` (plano de datos) y el segundo para la elaboración de señalización proveniente del módulo `executor` o de eventos de red generados en el gestor de estados (ver 1.1.7.2) (plano de control y gestión).

En el se implementa todas las funciones de encaminamiento que son necesarias transmitir a los controladores pues estos no conocen nada del plano cognitivo, son meras palancas de acción. En este módulo también implementamos el submódulo `Access` encargado de compartir con el resto de nodos parte de nuestro repositorio que es enviado por el canal `VCC`.

El diseño del canal `VCC` encargado de transmitir toda la información de control y gestión de la CWSN, se hubica en la misma red por la que circulan los datos, a diferencia de otros sistemas como el SS7 (*Signalling System no. 7*) [3] por razones de simplicidad. De hecho comparten ambos tráficos el mismo canal, la manera de distinguirlos reside en el valor de la etiqueta `MessageType`. Sin embargo la arquitectura no cierra las puertas a que el plano de datos y el de control estén sean transmitidos en distintas interfaces de comunicación, por ejemplo datos en *WiFi* y señalización en *Bluetooth*.

1.1.7. Interfaz Módulo Cognitivo - Controlador

1.1.7.1. Comunicación hacia controlador: invocación de métodos

La forma de comunicarse con el controlador de la interfaz es mediante la invocación de métodos del propio controlador. Al basarse en una programación en objetos, el módulo o servicio cognitivo tiene una referencia a cada controlador que le permite invocar sus métodos declarados como públicos permitiendo disparar procesos, para homogeneizar y que el servicio interactue de forma similar con todas las interfaces, sus controladores deben implementar una serie de métodos que desencadenen un proceso que al final tenga una respuesta parecida. Estos métodos son asíncronos cuya llamada no devuelve ningún valor, es a través de la interfaz entre las capas pero en el otro sentido como obtenemos la resuesta. Entre la serie de métodos que un controlador debe implementar, los indispensables al ser comunes a todos son:

start()
: Este método nos permite configurar el controlador teniendo en cuenta los requerimientos que tenga el módulo o servicio cognitivo.

stop()
Gracias a él se liberan todos los recursos (menos los relacionados con el módulo `discovery`).

write()
Para escribir cualquier tipo de información sobre la red:

- 'Plano de datos': canal de datos.
- 'Plano de control': canal `VCC`.

1.1.7.2. Comunicación hacia módulo: Gestor de estados

O *State Manager*, es una interfaz de comunicación uniderecional desde los distintos controladores hacia el servicio para expresar el estado en el que se encuentran éstas. Estos mensajes son consecuencia de procesos bien disparados por el servicio al invocar métodos del controlador (módulo ejecutor e `Interface Manager`) o bien originados en el propio controlador al tratarse de un evento de red.

WiFi Los diferentes estados y eventos en los que pueden estar la interfaz *WiFi*:

WiFi State Manager

Estados interfaz	Eventos interfaz
STATE_NONE	EVENT_NO_DEVICE_CHANGES
STATE_LISTEN	EVENT_ENTERED_IN_NETWORK
STATE_SENDING	EVENT_LOST_NODE
STATE_RECEIVING	EVENT_HELLO_NOT_REACHED

Tabla 1.1: Gestor de estados WiFi

Bluetooth De manera análoga los diferentes estados y eventos en los que pueden estar la interfaz *Bluetooth*:

Bluetooth State Manager

Estados interfaz	Eventos interfaz
STATE_NONE	EVENT_NO_DEVICE_CHANGES
STATE_LISTEN	EVENT_NEW_DEVICE
STATE_CONNECTING	EVENT_DEVICE_LOST
STATE_CONNECTED	EVENT_CONNECTING_DEVICE_FAILED
STATE_SENDING	EVENT_CONNECTION_ALREADY_EXISTING
STATE_RECEIVING	

Tabla 1.2: Gestor de estados Bluetooth

Esta información recibida por cada controlador se mapea con los eventos y estados que veremos en la API (ver sección 1.2.5) de forma que las singularidades de cada interfaz sean ocultadas más allá del servicio.

1.2. API: Definición, Cómo montar una aplicación sobre el servicio cognitivo

En esta sección se explica cada detalle de la interfaz Aplicación - Módulo Cognitivo en ambos sentidos, su cumplimiento es necesario para desarollar una aplicación que interactue de forma correcta con el módulo o servicio cognitivo. La forma de interactuar con éste se basa en, como se verá en 2.1, el intercambio de mensajes a través de los objetos *Messenger* y *Handler* provistos por *Android*. Estos mensajes provocan el disparo de ciertos procesos en el servicio que responderá, si procede, en cuanto el resultado esté listo. Son llamadas por tanto asíncronas, de igual modo el servicio entregará mensajes a las aplicaciones en cualquier momento sin que estás hayan solicitado nada.

La aplicación que haga uso del modelo no necesita cumplir más requisitos que la de manejar los distintos mensajes que le entrege el servicio en cualquier instante de tiempo y la de almacenar la lista de nodos formada por el par (identificador, nombre) entregada por el servicio y modificada en cada evento que lo requiera. La confección de algunos mensajes de la API requieren que sean informados con algunos valores de la lista de nodos, por lo que la aplicación debe tener accesible esta lista ya que no hay una manera activa de pedirla al servicio.

Los mensajes que se intercambian tienen un campo *what* que resume el proposito de éste, más una serie de añadidos experesados en forma de parámetros y extras que conforman y describen el proceso que está ocurriendo. Este campo *what* es un entero único que define el número diferente de mensajes tipo que se intercambian, entre éstos habrá algunos que sólo existan en un sentido de la comunicación y otros que aparezcan en ambos sentidos.

Mensajes tipo intercambiados entre Aplicación y servicio

Constante usada en el código	Representación en forma de entero
REGISTER_CLIENT	1
UNREGISTER_CLIENT	2
REGISTER_EXCHANGE	3
OUTGOING_CONTENT	4
OUTGOING_CONTENT_PROGRESS	5 | no implementado
OUTGOING_CONTENT_RESULT	6
INCOMING_CONTENT_PROGRESS	7 | no implementado
INCOMING_CONTENT	8
SERVICE_INFORMATION	9
ERROR_MESSAGE	10
QUEUED_MESSAGE	11
CONNECT_TO_VIA_BT	12
WORKAROUND_WHAT_TO_REPLY	20
WORKAROUND_ASK_RESPONSE	21
WORKAROUND_CONTEXT_CHANGE	22
WORKAROUND_PENDINGMESSAGE_INJECTION	23
WORKAROUND_PENDINGMESSAGE_DISPOSAL	24

Tabla 1.3: Mensaje tipo

Las filas en gris corresponden a mensajes reservados no implementados como son las constantes `OUTGOING_CONTENT_PROGRESS` (5) y `INCOMING_CONTENT_PROGRESS` (7). Están pensadas para una posible segmentación de los mensajes de aplicación por parte del servicio. En caso de segmentación la forma de informar a la aplicación del progreso del envío de segmentos (y de la recepción de éstos) se haría con este tipo de mensajes. Las filas precedidas por `WORKAROUND` son comportamientos necesarios para disparar procesos manualmente, corresponden por tanto a una interfaz de *test* no necesario su cumplimiento para el correcto funcionamiento.

1.2.1. Conectarse al servicio

Para poder entregar y recibir mensajes necesitamos una referencia al *Messenger* del servicio, esta referencia la obtenemos tras hacer una petición de *bind* llamando al método *bindService()* con un *Intent* a la clase:

`es.upm.die.lsi.pfc.CWSNoA.CognitiveLayer_specific_Service.class`

En la petición de *bind* también se pasa como parámetro un objeto de la clase *ServiceConnection* que el sistema *Android* llama cuando ha establecido la conexión. *Android* arrancará el servicio si éste no lo estaba ya y nos devolverá como objeto *IBinder* una instancia del *Messenger* del servicio. Por motivos de seguridad la aplicación que realice la petición de *bind()* requiere la declaración en su archivo '*manifest*' la aceptación del permiso:

`es.upm.die.lsi.pfc.CWSNoA.permission.COGNITIVE_SERVICE`

Este permiso informa al usuario final que usa la aplicación del uso posible de la modificación de la interfaz *Bluetooth* y *WiFi* sin la intervención de éste.

Una vez recuperada la referencia procedemos al registro de la aplicación en el servicio, para ello iniciamos un proceso de *handshaking*. El primer mensaje no lleva ningún dato adicional, dicta de esta forma (ver tabla 1.4).

Campos mensaje desde la aplicación al servicio				
What	*arg1*	*arg2*	*obj*	*replyTo*
1 = REGISTER_CLIENT	No usado/indiferente	No usado/indiferente	No usado/indiferente	Messenger de la aplicación

Tabla 1.4: Mensaje registro cliente

Acto seguido debemos informar sobre parámetros de nuestra aplicación al servicio con el envío de un mensaje con la siguiente información:

Papel de nodo En el campo *arg1* del mensaje. '0' para indicar que somos un nodo secundario y '1' para indicar que nuestro papel es primario.

Código de aplicación Como extra del mensaje incluimos una *cadena de caracteres* con la clave "appCode" indicando qué aplicación somos, para intercambiar mensajes cuyo código de aplicación coincida.

A través del mensaje descrito en la tabla 1.5

Tras esto el servicio nos responderá con un mensaje igual tipo `REGISTER_EXCHANGE`, pero del que puede haber dos variantes dependiendo de cuál fuera la situación del

Campos mensaje desde la aplicación al servicio				
What	*arg1*	*arg2*	*obj*	*replyTo*
3 = REGISTER_EXCHANGE	0=secundario / 1=primario	No usado/indiferente	*null*	Messenger de la aplicación
Extras				
key	*value*			
appCode	Una string con el código de aplicación			

Tabla 1.5: Mensaje entrega parámetros de aplicación

servicio. La forma fácil de identificar el tipo de respuesta es con la información devuelta en el campo *obj* del mensaje.

Se trata de un valor booleano, si obtenemos un '*false*' corresponde al caso de que no estabamos cooperando en una CWSN cuando nos registramos en el servicio. Es probable que nos hayan llegado entre tanto otros mensajes del tipo `SERVICE_INFORMATION` debido a procesos internos del servicio relacionados con la configuración y la entrada en la red. El detalle de éste se puede ver en la tabla 1.6

Campos mensaje desde el servicio a todas las aplicaciones				
What	*arg1*	*arg2*	*obj*	*replyTo*
3 = REGISTER_EXCHANGE	Estado de la interfaz	Interfaz	**False**	No usado/indiferente
Extras				
key	*value*			
nodeRole	Papel del nodo			
nodeType	Tipo de nodo			
periodTask	Periodo tarea cognitiva			
nodeName	Nombre del nodo en la red			
errorSetupInterface	Resultado configuración interfaz			

Tabla 1.6: Mensaje finalización de *handshaking*: CWSN no establecida

La información de cada campo:

Arg1: Estado de la interfaz Un entero cuyos valores corresponden a:

0. `INTERFACE_STATE_DOWN`, la interfaz de comunicación no está configurada.
1. `INTERFACE_STATE_IDLE`, la interfaz está configurada pero no estamos cooperando en red.
2. `INTERFACE_STATE_IDLE_NETWORKING`, situación de reposo cooperando en CWSN.
3. `INTERFACE_STATE_CONNECTING`, nos estamos conectando a otro nodo (sólo en *Bluetooth*).
4. `INTERFACE_STATE_SENDING`, estamos enviado datos a través de la interfaz.
5. `INTERFACE_STATE_RECEIVING`, estamos reciviendo datos.

Arg2: Interfaz Un entero cuyos valores corresponden a:

0. *Bluetooth*.

1. *WiFi*.
2. *Mobile*, no implementado.

-1. Desconocido, devuelto en el transcurso de cambio de contexto o *handover*.

Obj: '*false*' Un valor booleano cuyo valor nos indica que no estabamos cooperando con otros nodos tras la recepción del mensaje `REGISTER_EXCHANGE` por lo tanto el servicio ha configurado de nuevo la interfaz para disparar los procesos de entrada en la red (ver sección 1.3.2).

EXTRAS Campos *extra* con el resto de parametros cuando el valor *obj* es '*false*':

Papel del nodo Un entero que representa el papel del nodo en la red, se recupera con la clave "nodeRole":

0. Secundario.
1. Primario.

Tipo de nodo Un entero que representa el tipo de nodo, se recupera con la clave "nodeType":

0. Normal.
1. Coordinador.
2. Coordinador temporal en *Bluetooth*, no implementado. En todo caso este valor no es selecionable nunca por el usuario/servicio.

-1. Desconocido, valor por defecto (si obtenemos este valor nos encontramos en situación de error).

Periodo tarea cognitiva Un *double* con el que se nos informa cada cuánto el servicio realiza la tarea cognitiva (sensado, toma de decisiones...). Se recupera con la clave "periodTask".

Nombre del nodo en la red Una *cadena de caracteres* con la que el servicio nos informa de cómo nos conocen los otros nodos en la red. Se recupera con la clave "nodeName".

Resultado configuración interfaz Un valor booleano cuya clave para obtenerlo es "errorSetupInterface" cuyo valor significa:

- '*true*' se ha producido un error mientras la configuración de la interfaz.
- '*false*' todo ha ido OK en el proceso de configuración.

Si por el contrario el valor obtenido en el campo *obj* es '*true*' significa que el servicio ya se encontraba cooperando en una CWSN cuando registramos nuestra aplicación por lo que no es necesario ninguna configuración y recibimos una respuesta casi inmediata (a diferencia del caso anterior) con la siguente disposición (ver tabla 1.7).

La información recibida es prácticamente igual salvo el campo *obj* que nos devuelve el valor '*true*' lo que provoca que este mensaje sólo lo reciba esta aplicación en exlcusiva y un cambio en los extras, que dejan de inlcuir el 'Resultado configuración interfaz' pero incluyen dos nuevos:

Código de la aplicación Un *cadena de caracteres* que representa el código de la aplicación. El servicio nos hace un 'eco' de este valor para saber que se ha configurado bien. Se recupera con la clave "appCode".

Campos mensaje desde el servicio a la aplicación				
What	*arg1*	*arg2*	*obj*	*replyTo*
3 = REGISTER_EXCHANGE	Estado de la interfaz	Interfaz	**True**	No usado/indiferente
Extras				
key	*value*			
nodeRole	Papel del nodo			
nodeType	Tipo de nodo			
periodTask	Periodo tarea cognitiva			
nodeName	Nombre del nodo en la red			
appCode	Código de la aplicación			
nodeIdList	Array de enteros con los identificadores de los nodos			
nodeNameList	Array de *cadena de caracteres* con los nombres de los nodos			

Tabla 1.7: Mensaje finalización de *handshaking*: CWSN establecida

Lista de nodos Un conjunto de dos *arrays* ligados, uno con los identificadores de los nodos y otro con los nombres de éstos. De tal forma que el par (identificador, nombre) comparta la misma posición en ambos *arrays*. Se recuperan con las claves "nodeIdsList" y "nodeNamesList" respectivamente.

1.2.2. Enviar datos

El envío de datos a través de la red cognitiva consiste en la entrega al servicio de un mensaje codificado con esta información:

Payload El contenido del mensaje lo adjuntamos en el campo *obj* del mensaje, debe ser o poder codificarse como una *cadena de caracteres*.

Lista nodos Los nodos a quién va dirigido el mensaje se codifican como una lista de identificadores, instancia de un *ArrayList<Integer>* y debe ir como un extra guardado con la clave "addressedNodes". El identificador del nodo nos los devuelve el propio servicio en mensajes tipo `REGISTER_EXCHANGE` en situación de *networking* y en mensajes tipo `SERVICE_INFORMATION`. Como convención, el identificador es un número natural. El valor '0' puede ser utilizado por el programador de aplicaciones como ausencia de nodos (útil para rellenar *listviews* o *spinners*), el '1' es un valor reservado por el servicio que sirve para enviar el mensaje a todos los nodos. Los demás números corresponden cada uno a un nodo distinto.

Aunque no hay restricciones la manera correcta de enviar el mensaje a todos es rellenar la lista sólo con el identificador '1'.

El detalle del mensaje enviado al servicio es (ver tabla 1.8).

1.2.3. Confirmación envío de datos

Cuando hemos enviado datos a través del servicio, la forma que tiene éste de informarnos que el proceso de envío ha terminado es con la entrega del siguiente mensaje, del que podremos obtener las estadísticas de envío (ver tabla 1.9).

La información que podemos leer de este mensaje es:

Campos mensaje desde la aplicación al servicio				
What	*arg1*	*arg2*	*obj*	*replyTo*
4 = OUTGOING_CONTENT	No usado/indiferente	No usado/indiferente	*payload*	Messenger de la aplicación
Extras				
key	*value*			
addressedNodes	Una lista de enteros con los identificadores de los nodos destinatarios del mensaje			

Tabla 1.8: Mensaje para el envío de datos

Campos mensaje desde el servicio a la aplicación				
What	*arg1*	*arg2*	*obj*	*replyTo*
6 = OUTGOING_CONTENT_RESULT	número mensajes escritos OK	número mensajes escritos NOK	*payload*	No usado/indiferente
Extras				
key	*value*			
IdsNotDelivered	Un array de enteros con los identificadores de los nodos a los cuales no les ha llegado el mensaje			
NamesNotDelivered	Un array de *cadena de caracteres* con los nombres de los nodos a los cuales no les ha llegado el mensaje			

Tabla 1.9: Mensaje resultado envío de datos presentado a la aplicación

Arg1: número mensajes escritos OK Un entero que nos indica en cuántos nodos se ha entregado el mensaje correctamente.

Arg2: número mensajes escritos NOK Un entero con el número de nodos los cuales no han recibido el mensaje.

Obj: payload El contenido que se ha enviado por la red, esto sirve a la aplicación para relacionar el envío con la contestación.

EXTRAS Si el campo *arg2* es distinto de cero incluimos los siguientes extras:

Lista de nodos a los cuales no se han entregado el mensaje Un conjunto de dos *arrays* ligados, uno con los identificadores de los nodos y otro con los nombres de éstos. De tal forma que el par (identificador, nombre) comparta la misma posición en ambos *arrays*. Se recuperan con las claves "nodeIdsList" y "nodeNamesList" respectivamente.

La motivación de enviar tanto el identificador como el nombre es porque si hay algún fallo en la entrega del mensaje antes de la confirmación de entrega, habremos recibido un mensaje indicándonos la pérdida de algún nodo. La aplicación entonces puede haber borrado de su lista el nodo implicado que al intentar recuperar ahora (en la recepción de la confirmación) el nombre le dará error al no encontrarse ya en su lista.

El tiempo de entrega de esta confirmación no está acotado, depende de la complejidad en el envío y de si ha habido encolamiento en cualquiera de los segmentos de red

por los que ha pasado el mensaje.

1.2.4. Recibiendo datos

En cuanto el servicio haya recibido datos que le interesen a nuestra aplicación (es decir el código de aplcación sea el mismo con el que va marcado el mensaje de datos) nos enviará un mensaje con el contenido (en el campo *obj* del mensaje) y el identificador del nodo que lo envió, un extra que se recupera con la clave "device_id" (ver tabla 1.10).

Campos mensaje desde el servicio a la aplicación

What	*arg1*	*arg2*	*obj*	*replyTo*
8 = INCOMING_CONTENT	No usado/indiferente	No usado/indiferente	*payload*	No usado/indiferente

Extras

key	*value*
device_id	Un entero que identifica al nodo que ha enviado el mensaje

Tabla 1.10: Mensaje datos recibidos

1.2.5. Recibir información del servicio

El servicio utiliza esta vía como principal para informar acerca de la evolución de sus procesos internos. Gracias a su gestor de estados entre los controladores de las interfaces y éste, los eventos y estados se codifican y aglutinan para informar a todas las aplicaciones registradas acerca de éstos y cuándo se producen.

Aunque parte de esta información (estado de la interfaz y lista de nodos) se distribuya también en otros mensajes, es aquí dónde se plasman todos los cambios de estado que haya a lo largo de un proceso completo. En otros mensajes enviados por el servicio la información aportada tan sólo corresponde al momento final del proceso.

Diferentes estados en los que se pueden encontrar las interfaces

Constante usada en el código	Representación en forma de entero
INTERFACE_STATE_DOWN	0
INTERFACE_STATE_IDLE	1
INTERFACE_STATE_IDLE_NETWORKING	2
INTERFACE_STATE_CONNECTING	3
INTERFACE_STATE_SENDING	4
INTERFACE_STATE_RECEIVING	5

Tabla 1.11: Estados de la interfaz de comunicación

Los eventos se producen entre los dos nodos invulcrados directamente en el proceso. Por ejemplo, la inclusión de un nuevo nodo produce un evento en los controladores de la interfaz de los dos nodos implicados, el resto de nodos se enteran del evento con mensajes de señalización (que veremos en la sección 1.3) que para los controladores son tratados de la misma manera que otros mensajes de red. La mayoría de eventos surgen

del propio controlador de la interfaz, otros sin embargo son fruto de la interpretación de mensajes de señalización como son los cambios de contexto.

Eventos producidos en la red

Constante usada en el código	Representación en forma de entero
EVENT_INTERFACE_NO_CHANGES	0
EVENT_INTERFACE_NEW_NODE	1
EVENT_INTERFACE_LOST_NODE	2
EVENT_INTERFACE_CANNOT_CONNECT	3
EVENT_INTERFACE_ALREADY_CONNECTED	4
EVENT_INTERFACE_SWITCH_TO_BT	5
EVENT_INTERFACE_SWITCH_TO_WIFI	6
EVENT_INTERFACE_SWITCH_TO_MOBILE	7 | no implementado
EVENT_INTERFACE_END_SWITCHING	8
EVENT_UPDATE_LIST_NODES	9

Tabla 1.12: Eventos que se pueden producir en la red

Estos mensajes son puramente informativos, no requieren de ningún procesamiento por parte de las aplicaciones y se presentan con esta estructura (ver tabla 1.13).

Campos mensaje desde el servicio a todas las aplicaciones

What	*arg1*	*arg2*	*obj*	*replyTo*
9 = SERVICE_INFORMATION	interfaz	No usado/indiferente	modificación lista nodos	No usado/indiferente

Extras

key	*value*
coordinator	Una *cadena de caracteres* con el nombre del nodo que es nuestro coordinador. Si somos el cooridnador este campo viene informado a *null*
triggerNodeName	Una *cadena de caracteres* con el nombre del nodo que ha disparado el evento. Si no existe tal nodo no se incluye este campo en el mensaje
state	Una entero con el estado de la interfaz
event	Una entero con el evento producido en la red
nodeIdList	Un array de enteros con los identificadores de los nodos que forman la red en estos momentos
nodeNameList	Un array de *cadena de caracteres* con los nombres de los nodos que forman la población actual

Tabla 1.13: Mensaje información evento/estado por parte del servicio

El detalle de cada uno de los campos es:

Arg1: Interfaz Un entero cuyos valores corresponden a:

0. *Bluetooth*.
1. *WiFi*.
2. *Mobile*, no implementado.

-1. Desconocido, devuelto en el transcurso de cambio de contexto o *handover*.

Obj: modificación lista nodos Un valor booleano que nos indica si ha habido un cambio en la lista de nodos.

EXTRAS Campos *extra* con el resto de parámetros:

Coordinador Una *cadena de caracteres* con el nombre del nodo coordinador. Informado a *null* si somos nosotros el coordinador de la red.

Nodo causante del evento Una *cadena de caracteres* con el nombre del nodo que ha causado el evento. Esta información es más útil para el nodo coordinador, pues para el normal el nodo causante siempre será su coordinador.

Estado interfaz Un entero cuyos valores son los vistos en la tabla 1.11, se obtiene con la clave "state":

0. `INTERFACE_STATE_DOWN`, la interfaz de comunicación no está configurada.
1. `INTERFACE_STATE_IDLE`, la interfaz está configurada pero no estamos cooperando en red.
2. `INTERFACE_STATE_IDLE_NETWORKING`, situación de reposo cooperando en CWSN.
3. `INTERFACE_STATE_CONNECTING`, nos estamos conectando a otro nodo (sólo en *Bluetooth*).
4. `INTERFACE_STATE_SENDING`, estamos enviado datos a través de la interfaz.
5. `INTERFACE_STATE_RECEIVING`, estamos reciviendo datos.

Evento producido Un entero cuyos valores son los vistos en la tabla 1.12, se recupera con la clave "event":

0. `EVENT_INTERFACE_NO_CHANGES`, este evento es la ausencia de evento. Hay cambios en el estado de la interfaz que no son provocados por un evento, para estas casuisticas el evento que le asignamos es el de 'no cambio'.
1. `EVENT_INTERFACE_NEW_NODE`, este evento surge cuando el coordinador nos conecta a la red. Se genera en ambos extremos (en el coordinador y el nodo normal que empieza a formar parte de la red).
2. `EVENT_INTERFACE_LOST_NODE`, se genera al perder la comunicación con un nodo (entre coordinador y normal, aparece este evento al igual que el anterior en ambos extremos).
3. `EVENT_INTERFACE_CANNOT_CONNECT`, surge en el nodo que ha intentado conectarse a otro y no ha podido.
4. `EVENT_INTERFACE_ALREADY_CONNECTED`, surge cuando el coordinador intenta conectarse a un nodo al cual ya está conectado (sólo en *Bluetooth*).
5. `EVENT_INTERFACE_SWITCH_TO_BT`, evento que surge cuando recibimos o disparamos (coordinador) el cambio de contexto hacia *Bluetooth*.
6. `EVENT_INTERFACE_SWITCH_TO_WIFI`, evento que surge cuando recibimos o disparamos (coordinador) el cambio de contexto hacia *WiFi*.
7. `EVENT_INTERFACE_SWITCH_TO_MOBILE`, no implementado, de manera análoga cuando la interfaz de destino es la móvil.
8. `EVENT_INTERFACE_END_SWITCHING`, se genera al terminar el cambio de contexto.
9. `EVENT_UPDATE_LIST_NODES`, cuando es necesaria una actualización de la lista de nodos que no haya sido provocada por algún evento anterior.

Lista de nodos Un conjunto de dos *arrays* ligados, uno con los identificadores de los nodos y otro con los nombres de éstos. De tal forma que el par (identificador, nombre) comparta la misma posición en ambos *arrays*. Se recuperan con las claves "nodeIdsList" y "nodeNamesList" respectivamente.

1.2.6. No cumplimiento de la API: mensajes de error

El servicio tiene una manera de avisar a las aplicaciones cuando éstas no cumplen con la API o internamente surge un error cuyo resultado es conveniente comunicar para poner en conocimiento que el proceso actual se ha detenido. Este aviso se realiza mediante el envío de mensajes de error que están dirigidos a quién lo ha provocado, o puesto en conocimiento de todos según interese.

Es una sección experimental que sólo recoge los errores más importantes. No se ha hecho un estudio exhaustivo sobre el comportamiento maligno de las aplicaciones para intentar manejar y sanitizar todos los valores entregados por éstas. Algunos de los errores de los que son manejados por el servicio se listan en la tabla 1.14

Mensajes de error devueltos por el servicio

Constante usada en el código	Representación en forma de entero
CODE_ERROR_ILLEGAL_ARGUMENT	1
CODE_ERROR_NODE_NO_EXISTS	2
CODE_ERROR_INTERNAL	3
CODE_ERROR_NOT_IN_NETWORK	4
CODE_ERROR_MESSAGE_NOT_RECOGNIZED	5
CODE_ERROR_ILLEGAL_PAYLOAD	6

Tabla 1.14: Mensajes de error por el no cumplimiento de la API

Los errores son transmitidos a la aplicación o las apliaciones por medio de este tipo de mensaje (ver tabla 1.15), en él se amplia la descripción del error mediante una *cadena de caracteres* en el campo *obj*.

Campos mensaje desde el servicio a la aplicación

What	*arg1*	*arg2*	*obj*	*replyTo*
10 = ERROR_MESSAGE	código de error	No usado/indiferente	Descripción del error	No usado/indiferente

Tabla 1.15: Mensaje de error recibido del servicio

1.2.7. Mensajes encolados

Puede darse la circustancia de que al realizar el envío de datos al servicio, éste no pueda enviarlo por la red debido a que se encuentre en un cambio de contexto. En estas ocasiones los mensajes recibidos por parte de las aplicaciones son encolados siguiendo el modelo FIFO (*First-in First-out*), esto quiere decir que se respeta el orden de llegada y serán entregados en el mismo orden cuando se proceda su salida.

Cuando el servicio retoma el envío de estos mensajes encolados, envíamos a la aplicación dueña del mensaje un informe previo a su envío, indicando cuáles de los nodos destinatarios siguen en la red y cuáles no. En el caso de haber enviado el mensaje a todos, este informe nos devuelve la lista completa de nodos a los cuales le hemos intentado enviar el mensaje, ya que puede haber cambios en la población entre el momento en el cual se envío y el momento en el que el servicio le da salida. El detalle del mensaje recibido en la aplicación es (ver tabla 1.16).

Campos mensaje desde el servicio a la aplicación

What	*arg1*	*arg2*	*obj*	*replyTo*
11= QUEUED_MESSAGE	número enviados	número no enviados	*payload*	No usado/indiferente

Extras

key	*value*
IdsTriedToSend	Un *array* de enteros con los identificadores de los nodos los cuales siguen en red
NamesTriedToSend	Un *array* de *cadena de caracteres* con los nombres de los nodos siguen en red
IdsNotDelivered	Un *array* de enteros con los identificadores de los nodos que no hemos enviado el mensaje por no encontrarse en la red
NamesNotDelivered	Un *array* de *cadena de caracteres* con los nombres de los nodos que no hemos enviado el mensaje por no encontrarse en la red

Tabla 1.16: Mensaje informe previo a la salida de mensaje encolado

Los campos *extra* son dos pares de *arrays* ligados por la posición al igual que ocurre con la lista de nodos de otros mensajes. Los dos primeros nos informan de cúales son los nodos a los cuales se va a proceder al envío del mensaje y los dos últimos están rellenos de los nodos que al no encontrarse en este momento en situación de red hemos obviado su envío para evitar envíos fallidos innecesarios.

Al recibir este mensaje quedamos a la recepción del mensaje de confirmación de envío explicado en 1.2.3

1.2.8. Actualización de parámetros

Si en algún momento queremos cambiar nuestro papel en la red o recibir la información de otras aplicaciones de la red cognitiva tan sólo necesitamos cumplir cierta parte del flujo del *handshaking* para conectarse al servicio (ver 1.2.1). Empezaremos enviando directamente el segundo mensaje, con nuestos nuevos parámetros al servicio con el mensaje ya visto en la tabla 1.5, tras esto esperaremos la respuesta dónde veremos si realmente se han podido actualizar los valores que les entregamos. Si cuando decidimos actualizar estabamos en situación de *networking* es posible que los valores no hayan podido actualizarse debido a que penalizan o entran en conflicto con las necesidades de las otras aplicaciones ya registradas. En este caso sólo se permite un cambio de papel 'secundario' a 'primario', pero no al revés. El 'código de aplicación' se actualiza siempre no importando la situación.

1.2.9. Desconectarse del servicio

Llegado el momento de desconectarnos de la red deberemos enviar al servicio un mensaje del tipo *unregister client* para que el servicio sea informado de forma directa, ya que la mera petición de *unbind* no garantiza que éste se entere y si se entera, es para su propia destrucción por parte de *Android*. La composición de este mensaje es lo podemos ver en la tabla 1.17

Campos mensaje desde la aplicación al servicio				
What	*arg1*	*arg2*	*obj*	*replyTo*
2 = UNREGISTER _CLIENT	No usado/indiferente	No usado/indiferente	No usado/indiferente	Messenger de la aplicación

Tabla 1.17: Mensaje desregistro cliente

Tras este envío ya podemos deshacer el enlace entre aplicación y servicio de manera segura.

1.2.10. Entrada de nodos en la red cuando la interfaz es *Bluetooth*

Por motivos de implementación, si queremos introducir un nuevo nodo en la red cuando la interfaz de comunicación es *Bluetooth*, debe ser el coordinador quién de el primer paso. Para ello en el nodo coordinador enviamos el siguente mensaje (ver tabla 1.18).

Campos mensaje desde la aplicación al servicio				
What	*arg1*	*arg2*	*obj*	*replyTo*
12 = CONNECT_TO _VIA_BT	No usado/indiferente	No usado/indiferente	dirección MAC / identificador nodo	Messenger de la aplicación

Tabla 1.18: Mensaje contectar a nodo en *Bluetooth*

El campo *obj* puede recibir dos tipos de argumentos:

- Una *cadena de caracteres* con la dirección MAC de *Bluetooth* del nodo.
- Un entero con el identificador el nodo, si éste no está en la base de datos enviaremos un mensaje de error a la aplicación.

Acto seguido el servicio intentará si la interfaz de comunicación actual es *Bluetooth* conectarse al citado nodo bien por medio de su dirección MAC o porque hemos podido recuperar ésta de la base de datos. Si la interfaz no es *Bluetooth* devolveremos un error del tipo `CODE_ERROR_ILLEGAL_ARGUMENT`

Known issue El *HashMap* del controlador de *Bluetooth* donde guardamos las hebras de conexión pierde su contenido tras añadir un valor, la primera vez que se inicia el servicio. Instrucciones para detectar el fallo:

0. Vaciar la base de datos y matar el proceso. O instalar la aplicación.

1. Sin cerrar la aplicación (no destruir el servicio) intentar conectarse desde un coordinador a ésta.

2. Observar que cuando el nodo intenta responder al coordinador con su información o intenta enviar un escaneo se lanza un evento de pérdida por no poder recuperar del *HashMap* la hebra de conexión utilizada.

1.2.11. La API oculta: workarounds, test...

1.2.11.1. Actualización de parámetros avanzada

Un nodo cognitivo tiene mas parámetros (como se puede ver en 2.2.5) que los expuestos en la API, un nodo cognitivo en nuestra aplicación, se define además por:

- Nombre: con el que nos conocen los demas nodos.
- Tipo: si somos coordinador o normal.
- Periodo de la tarea cognitiva: el intervalo de tiempo que transcurre entre sensados del entorno y/o toma de decisiones.

Como vemos son parámetros que no son propios de la aplicación o aplicaciones que en ese momento estén registradas. Estos valores son devueltos cuando finaliza el *handshaking* para que las aplicaciones conozcan al nodo cognitivo aunque para éstas debe ser transparente.

Sin embargo para poder configurar el nodo para que pueda cooperar en cualquier CWSN debemos habilitar una manera de modificar estos parámetros. Bien, la forma es enviar el mismo mensaje que cuando actualizamos nuestro papel o código de aplicación pero con unos campos adicionales. En concreto el campo clave para saber si se trata de una actualización normal o avanzada recae en el campo *obj*. Vimos que en una actualización normal su valor es *null* y en una avanzada este valor será 'no nulo' como vemos en la estructura del mensaje (ver tabla 1.19). Los pormenores de este flujo los veremos en la sección 1.3

Campos mensaje desde la aplicación al servicio

What	*arg1*	*arg2*	*obj*	*replyTo*
3 = REGISTER_EXCHANGE	0=secundario / 1=primario	0=normal / 1=coordinador	**True**	Messenger de la aplicación

Extras

key	*value*
appCode	Una *cadena de caracteres* con el código de aplicación
nodeName	Una *cadena de caracteres* con el nuevo nombre del nodo en la red
periodTask	Un *double* con el nuevo periodo medido en segundos de la tarea cognitiva a ejecutar

Tabla 1.19: Actualización de todos los parámetros del servicio

1.2.11.2. Mensaje de petición y respuesta

Un flujo interesante en redes de sensores cognitivas es la de habilitar una vía para que el coordinador pida de forma activa información a otro nodo. Aunque en nuestra arquitectura este flujo está implementado la primera vez que incluimos un nodo nuevo en *Bluetooth* para preguntarle acerca de su nombre y su papel. Hemos preferido dejar montado, lo que en principio se montó con finalidad de *test*.

La aplicación desarrollada junto al servicio incluye una interfaz de test que, entre sus funciones provee este flujo. El proceso tiene dos actores: por un lado los nodos normales que pueden guardar una respuesta (la información que posteriormente nos pedirá el coordinador) para ello enviamos un mensaje al servicio con la respuesta a enviar en el campo *obj* del mensaje (ver tabla 1.20).

Campos mensaje desde la aplicación al servicio				
What	*arg1*	*arg2*	*obj*	*replyTo*
20 = WORKAROUND _WHAT_TO_REPLY	No usado/indiferente	No usado/indiferente	*reply*	Messenger de la aplicación

Tabla 1.20: Mensaje *workaround* salvado de respuesta

El otro agente, el nodo coordinador, enviando este mensaje al servicio recupera la información del otro actor (ver tabla 1.21).

Campos mensaje desde la aplicación al servicio				
What	*arg1*	*arg2*	*obj*	*replyTo*
21 = WORKAROUND _ASK_RESPONSE	No usado/indiferente	No usado/indiferente	Identificador nodo	Messenger de la aplicación

Tabla 1.21: Mensaje *workaround* petición de respuesta

Una vez enviada esta petición la respuesta nos vendrá encapsulada en un mensaje del mismo tipo (ver tabla 1.22). Esta respuesta se envía a todas las aplicaciones porque no viene marcada con el 'código de aplicación', se recomienda su uso sólo en test ya que puede desconcertar a otras aplicaciones que estén registradas en el servicio.

Campos mensaje desde el servicio a todas las aplicaciones				
What	*arg1*	*arg2*	*obj*	*replyTo*
21 = WORKAROUND _ASK_RESPONSE	No usado/indiferente	No usado/indiferente	Respuesta del nodo preguntado	Messenger de la aplicación

Tabla 1.22: Mensaje *workaround* respuesta obtenida

1.2.11.3. Cambio de contexto forzado

Otra función de *test* es la de forzar un cambio de interfaz de comunicación en la red. Para ello si somos el coordinador podemos enviar este mensaje al servicio. No

necesitamos comunicarle la interfaz destino debido a que, en estos momentos, sólo hay dos interfaces de comunicación.

Campos mensaje desde la aplicación al servicio				
What	*arg1*	*arg2*	*obj*	*replyTo*
22=WORKAROUND_ CONTEXT_CHANGE	No usado/indiferente	No usado/indiferente	No usado/indiferente	Messenger de la aplicación

Tabla 1.23: Mensaje *workaround* cambio de interfaz de comunicación

1.2.11.4. Encolar mensajes y forzar su entrega en cualquier momento

Para probar el encolamiento de mensajes y su entrega en momentos que no corresponden pueden enviarse estos mensajes al servcio que alterará el flujo normal para producir el resultado buscado.

Para encolar un mensaje de datos entregaremos al servicio un mensaje con esta estructura (ver tabla 1.24).

Campos mensaje desde la aplicación al servicio				
What	*arg1*	*arg2*	*obj*	*replyTo*
23=WORKAROUND_ PENDINGMESSAGE _INJECTION	No usado/indiferente	No usado/indiferente	*payload*	Messenger de la aplicación
Extras				
key	*value*			
addressedNodes	Una lista de enteros con los identificadores de los nodos destinatarios del mensaje			

Tabla 1.24: Mensaje *workaround* encolamiento de mensaje de datos

Y para vaciar la cola de mensajes en cualquier momento deberemos enviar este otro (ver tabla 1.25).

Campos mensaje desde la aplicación al servicio				
What	*arg1*	*arg2*	*obj*	*replyTo*
24=WORKAROUND_ PENDINGMESSAGE _DISPOSAL	No usado/indiferente	No usado/indiferente	No usado/indiferente	Messenger de la aplicación

Tabla 1.25: Mensaje *workaround* vaciado cola de mensajes

1.3. Procesos de red

Cuando ocurren eventos en la red estos son manejados por los distintos nodos según sea su tipo, propagando la información necesaria para que toda la red pueda reconstruir la misma perspectiva sobre la actual situación. Veremos quién dispara y el por qué de estos eventos, cómo se informa y en qué casos a las aplicaciones montadas sobre el servicio del nodo cognitivo. Esta sección viene muy relacionada con la sección

1.2 ya que dónde empieza o finaliza la acción es con mensajes intercambiados de la API.

1.3.1. Registro de una aplicación en el servicio cognitivo

Código invulcrado

Service: m2service (manejador de mensajes entre servicio y aplicaciones)

Para que una aplicación pueda registrarse de forma correcta en el servicio debe cumplir un cierto *handshaking* consistente en el intercambio de tres mensajes, el primero servirá para que el servicio incluya a la aplicación en su base de clientes. El segundo para que la aplicación le informe de sus parámetros y el tercero será una confirmación por parte del servicio hacia la aplicación o aplicaciones informando de todos los parámetros de éste tras la inclusión de la nueva aplicación.

Como hemos visto ya en la API (ver sección 1.2), tras la petición de *bind*, que inicializará el servicio si no estuviese arracando ya, obtendremos el *Messenger* del servicio, indispensable para poder comunicarnos con él. A continuación debemos mandar un primer mensaje para registrar nuestra aplicación en el servicio, donde éste, registrará nuestro messenger para habilitar la comunicación en sentido contrario y nos incluirá en su lista de aplicaciones registradas. Los detalles de este mensaje pueden verse en la tabla 1.4

El siguiente mensaje (ver tabla 1.5) que debemos enviar, informaremos acerca de nuestros parámetros de aplicación como son: nuestro papel, un entero cuyo valor '0' corresponde a papel secundario y el valor '1' corresponde al papel primario y nuestro código de aplicación, útil para que el servicio nos entregue sólo los mensajes que nos atañen.

Este mensaje ha sido separado del anterior para poder reutilizarlo e introducirlo en el flujo descrito en 1.2.8 o 2.2.5. Con esta información el servicio dependiendo del punto en que se encuentre actuará de una forma u otra.

1.3.1.1. Punto de partida inicial, CWSN no establecida

Esta situación, (la más común) es cuando nuestra petición de *bind* ha arrancado el servicio y por lo tanto estamos en una situación inicial o ya hay aplicaciones montadas sobre éste pero no hay conectividad con otros nodos, en otras palabras, estamos solos en la red. Un ejemplo cuando el servicio no está arrancado podemos verlo en la figura 1.2

En esta situación, configuraremos de nuevo la interfaz, actualizaremos los parámetros pasados en el anterior mensaje y lanzaremos nuevamente los procesos de registro en la red para intentar establecer una CWSN. Al finalizar éstos, el servicio devolverá (a todas las aplicaciones montadas) la información acerca de todos los parámetros y el estado actual de la red (ver listas de la sección 1.2.1). Puede verse el detalle de este mensaje en la tabla 1.6

1.3.1.2. Punto de partida, CWSN previamente configurada

Puede darse el caso que, en este punto del proceso, ya haya aplicaciones montadas sobre el servicio y estén cooperando en una CWSN. En este caso el servicio no tiene que configurar nada y se limita a ver si puede satisfacer las necesidades que le acaba de transmitir la nueva aplicación que acaba de registrar, es decir, cambiará el papel del

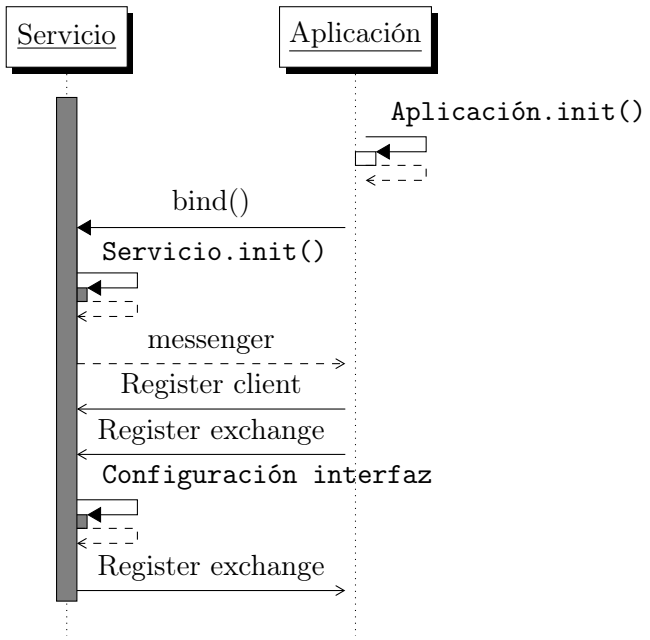

Figura 1.2: Handshake CWSN no configurada y servicio no inicilizado

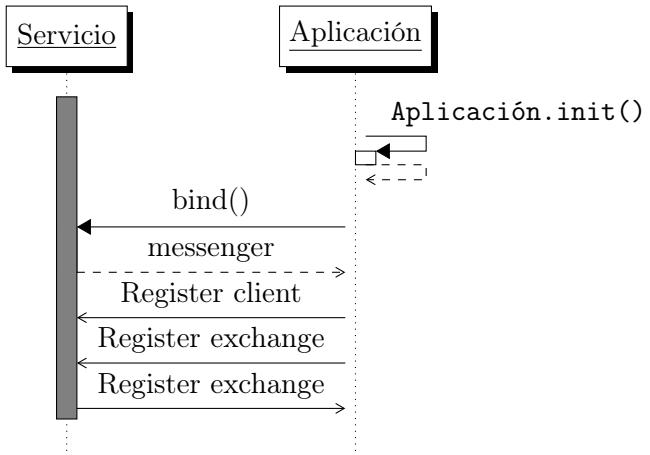

Figura 1.3: Handshake CWSN configurada

nodo a primario si este era secundario, un cambio en sentido contrario será ignorado. Una vez hecho esto devolverá en exclusiva a la aplicación solicitante los parámetros del servicio (ver tabla 1.29) y el estado de la interfaz. La estructura del mensaje devuelto puede apreciarse en la tabla 1.7

1.3.2. Configuración de la interfaz de comunicación

Tras recibir el *handshaking* de registro de la aplicación, el servicio si no se encuentra colaborando en una CWSN, procede a la configuración de una interfaz de comunicación que se realiza en segundo plano. Antes de empezar la configuración una interfaz pediremos a los controladores de las otras interfaces que liberen los recursos (llamando a su método *stop*) para partir de una situación con mínimos recursos utilizados.

1.3.2.1. WiFi: servicio y controlador

Código invulcrado

Service: SetupWifiInterface (AsyncTask)
WifiController: stop(), start() / StatusReceiver_wifi (BroadcastReceiver)

La interfaz *WiFi* permite configurarse en varias modalidades como: Infraestructura, Tethering o Ad-hoc. En la actualidad sólo está implementado el modo infraestructura por los problemas descritos en **??**. Una vez lanzada la tarea en segundo plano, intentamos liberar los recursos que vamos a solicitar si éstos estuvieran reservados por llamadas anteriores y no se hubiesen liberado aún.
La configuración comienza encendiendo la interfaz con la ayuda de la API de *Android*, si no lo estuviese ya. Esperamos a que se encienda mediante una espera de consulta activa o *polling*. Una vez levantada intentamos contectarnos a la red *WiFi* que nos hayan pasado como parámetro (mediante el SSID (*Service set identification*)), desconectándonos del punto de acceso actual si no fuese el mismo. Si no nos han pasado ningún parámetro dejamos al sistema *Android* que elija el AP (*Access point*) mas conveniente. Mientras se establece la conexión a un punto de acceso esperamos mediante *polling* con límite de tiempo seleccionable desde las preferencias en la sección de *test*. Si agotamos el tiempo y no hemos sido capaces de conectarnos, devolvemos un valor '*false*' que provoca el envío de mensaje a todas las aplicaciones del tipo `REGISTER_EXCHANGE` con el campo `EXTRA` "errorSetupInterface" a '*true*'. Si por el contrario nuestro *BroadcastReceiver* ha podido capturar la acción de conexión al citado punto de acceso o ya estamos conectados a éste continuamos estableciendo los canales de comunicación que dependen del tipo de nodo que seamos:

Normal Necesitamos tener un canal de comunicación permanentemente a la espera para tráfico TCP recibido desde el coordinador. Una vez establecido terminamos la configuración enviando un mensaje de tipo `REGISTER_EXCHANGE` con el campo `EXTRA` "errorSetupInterface" a '*false*' y el resto de campos descritos en la tabla 1.6 y lanzamos el proceso de registro en la red (sección 2.2.3).

Coordinador Necesitamos tener dos canales de comunicación, uno para tráfico proveniente de nodos ya en la red, este tráfico necesitamos que sea fiable por lo que será un canal TCP (*Transmission Control Protocol*) y otro canal para recibir avisos de nuevos nodos que se quieran incorporar a la red, este tráfico por motivos de implementación no puede ser fiable por lo que tendremos que recurrir a tráfico UDP (*User Datagram Protocol*). Al terminar de establecer los canales enviamos un mensaje análogo al párrafo anterior y quedamos a la espera de capturar paquetes.

1.3.2.2. Bluetooth: servicio y controlador

Código invulcrado

Service: SetupBTinterface (AsyncTask)
BluetoothController: stop(), start(), connect(), setState() / StatusReceiver_bt (BroadcastReceiver)

Como en la configuración del controlador de *WiFi* empezamos liberando si no estuviesen ya cada uno de los recursos. Encendemos la interfaz *Bluetooth* si no estuviese

encendida ya, mediante una petición al sistema *Android* y esperamos mediante *polling* a que finalice. A continuación, procedemos a configurar las hebras del controlador según el tipo de nodo:

Normal Por motivos de implementación este tipo de nodo se comporta como esclavo en la comunicación *Bluetooth* por lo que necesita aceptar conexiones, como sólo esperamos un maestro cuando se consigue una conexión liberamos este recurso de aceptación de conexiones. Como ya hemos configurado la interfaz enviamos el mensaje final del *handshaking* (ver tabla 1.6) con las instrucciones que hemos visto ya.

Coordinador Por motivos de implementacion este tipo de nodo se debe comportar como maestro en la comunicación *Bluetooth*, no necesita aceptar conexiones, pero si disparar éstas. Para ello, recogemos de la base de datos los nodos inactivos y para cada uno de ellos lanzamos una conexión gracias al método *connect()*. Este método coge como parámetros la dirección MAC con la cual queremos establecer una conexión y el número de intentos que vamos a intentar para establecer la conexión (a parte de otras constantes como el UIID).
Como en *Bluetooth* es fácil que falle el establecimiento, intentamos minimizar este riesgo mediante el reitero de la conexión. En el momento que conseguimos establecer el canal de comnunicación trasladamos este canal al manejador de conexiones del controlador que tendrá tantos canales como nodos haya en la red. Estos canales son cifrados y fiables por diseño de *Android*.
Cada vez que se produce o se falla en el establecimiento del canal, el controlador comunica al servicio esta información que es utilizada para saber cuando hemos de establecer el canal de comunicación con el siguente nodo.

Una vez hemos acabado con la lista de nodos inactivos habremos terminado de configurar la interfaz y procederemos al envío del último mensaje del *handshaking* con el formato visto en la tabla 1.6

1.3.3. Registro de un nodo en la red

Según la interfaz en la que nos encontremos se alterna el papel de quién da el primer paso. Así en *Bluetooth* el nodo normal espera una conexión entrante y en *WiFi* es el coordinador quién espera un paquete UDP enviado desde un nodo normal.

1.3.3.1. WiFi

Código invulcrado

WifiController: SendHelloPacket (AsyncTask), UDPlistener (Thread), stopSendingHelloPacket(), setState()
Service: mHandlerWf, sendInfoNewNodeOnNetwork()
Database: eventNewDeviceIPCoordinator(), newDeviceEventNormal(), modifyNode()

Tras la configuración de la interfaz por parte de un nodo normal, éste inicia otro proceso que comienza a enviar una ráfaga de paquetes espaciados en el tiempo, la razón radica en que al ser trafico *broadcast/multicast* tanto los puntos de acceso pueden menospreciarlo (incluso bloquearlo) como el mismo dispositivo *Android* puede que no esté escuchando siempre en este tipo de direcciones.

De hecho cuando se apaga la pantalla, el driver de *WiFi* se reconfigura automáticamente para dejar de escuchar a la dirección de *broadcast*, por ello una ráfaga espaciada en el tiempo aumenta las posibilidades de recepción e interpretación de estos paquetes por parte del coordinador. Si al finalizar el envío de la ráfaga no consiguiésemos ninguna respuesta el proceso de registro terminará enviando hacia el servicio el estado `EVENT_HELLO_NOT_REACHED` que será interpretado por éste informando a todas las aplicaciones registradas del fallo del proceso indicando que no ha sido posible establecer una CWSN, en este caso el evento facilitado a las aplicaciones se transforma en `EVENT_INTERFACE_CANNOT_CONNECT` que concentra el mismo tipo de evento en distintos controladores, uniformando la respuesta.

Este tipo de paquetes lo denominamos *"HelloPacket"* y su contenido es:

Estructura mensaje registro en la red *WiFi*

Tipo de paquete Valor "HELLO" del enumerado "packetType"

Hello packet Un contenedor que contiene al mensaje del tipo que se indicó en el campo "Tipo de paquete"

- **Nombre del nodo en la red** Una *cadena de caracteres* con el nombre del nodo en la red
- **Papel del nodo** Una *cadena de caracteres* cuyo valor 'p' es interpretado como primario y 's' como secundario ('u' para valor desconido)
- **Tipo de nodo** Una *cadena de caracteres* cuyos valores pueden ser: 'n' normal, 'c' coordinador ('t' coordinador temporal en *Bluetooth* [sin implementar], 'u' para valor desconocido)
- **Dirección MAC** Una *cadena de caracteres* cuyo valor es la dirección MAC de *Bluetooth*, clave que identifica al nodo de manera unívoca

Tabla 1.26: Campos del paquete "HelloPacket"

Si por el contrario uno de los paquetes llega a destino y es manejado por el coordinador, éste interpretará el contenido dentro del mismo controlador. Esta ruptura del modelo de capas tiene como objetivo evitar propagar información al servicio de paquetes que no encajen con la estructura y campos requeridos desechando otros paquetes. El controlador entiende como manejar estos paquetes pero no el resto. Si capturamos un *"HelloPacket"* generaremos un evento hacia el servicio de nuevo nodo en la red `EVENT_ENTERED_IN_NETWORK` con la información leída del paquete.

Este evento llega al *handler* de *WiFi* en el servicio, dónde lo primero que hacemos es ver si el nodo ya está registrado en la red (este paso es necesario ya que tenemos varios envíos del mismo paquete por parte del mismo nodo), si ya está registrado ignoramos este evento, pues es una copia de otro anterior durante una guarda de tiempo. La motivación de esta guarda recae en que en *WiFi* no se detecta inmediatamente que un nodo se ha caído de la red, sólo nos podemos dar cuenta cuando intentamos enviarle algo y obtenemos un error. Puede darse la situación de que un nodo se registre, se caiga e intente registrarse de nuevo. Si en este lapsus nadie ha intentado ponerse en contacto con él, el resto de nodos seguirán creyendo que el nodo está en la red, si ignorasemos este evento el nodo que está intentado cerrar su registro en la red no tiene

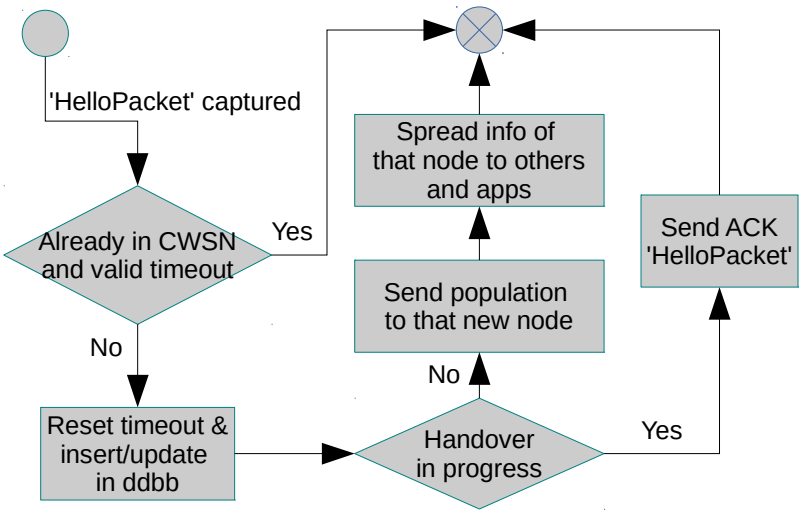

Figura 1.4: Flujograma de los procesos desencadenados en el coordinador al introducir un nodo en *WiFi*

manera de saber que en realidad a ojos de los demás nodos él ya está registrado en la red, creyendo que su *"HelloPacket"* no ha llegado a destino resultando una situación de falso fallo. Si no estaba activo en la red, lo marcamos como activo (insertándolo, si no estuviese ya, en la base de datos con la información facilitada por el controlador) Acto seguido el coordinador informará a éste nodo del resto de nodos que ya hay en la red y al resto de nodos de la red se les informará del nuevo nodo. Por su parte el coordinador informará a las aplicaciones registradas que ha habido un nuevo nuevo nodo en la red y facilitará la lista de nodos.

La información de señalización que se envía y que representa[1] al nodo la podemos ver en la tabla 1.27

Al recibir esta información el nodo normal, su controlador ya le habrá informado al servicio que está recibiendo datos, este evento le sirve al servicio para ver que ha sido registrado en la red y genera el mismo un evento de entrada en la red `EVENT_ENTERED_IN_NETWORK` que sirve para homogeneizar el flujo y que sea idéntico al de *Bluetooth*, este evento sirve para parar el envío de *"HelloPacket"*. Al terminar de recibir el mensaje de señalización antes de procesarlo, tendremos que comprobrar que el nodo coordinador ya está incluido en la base de datos y si no incluir su dirección MAC para obtener el identificador de nodo. El esquema de base de datos nos exige conocer a priori el identificador del nodo para poder traducir las direcciones MAC de *Bluetooth*. A medida que vamos procesando el mensaje vamos sustituyendo los valores anteriores (o de relleno si el nodo coordinador no estaba incluido en la base de datos) por los nuevos extraidos del mensajes de señalización. En este momento el nodo normal informará a las aplicaciones de que ha entrado en la red gracias a mensajes vistos en 1.2.5 con el evento `EVENT_INTERFACE_NEW_NODE` (visto en 1.12) y anunciará la lista de nodos quién es el coordinador.

[1]Notar que el campo "Dirección IP" puede no ser informado por no estar disponible en ese momento o ser inecesario

Estructura mensaje representación de un Nodo (serialización)

Nombre del nodo en la red Una *cadena de caracteres* con el nombre del nodo en la red.

Papel del nodo Una *cadena de caracteres* cuyo valor 'p' es interpretado como usuario primario y 's' como secundario ('u' para valor desconocido).

Tipo de nodo Una *cadena de caracteres* cuyos valores pueden ser: 'n' normal, 'c' coordinador ('t' coordinador temporal en Bluetooth [sin implementar], 'u' para valor desconocido).

Dirección MAC Un *cadena de caracteres* cuyo valor es la dirección MAC en *Bluetooth* del nodo, clave que identifica al nodo de manera unívoca.

Dirección IP Una *cadena de caracteres* cuyo valor representa la dirección IP (formato IP4) del nodo.

Dirección MAC del coordinador Una *cadena de caracteres* que contiene la dirección MAC de *Bluetooth* del coordinador del nodo que estamos enviando la información.

Tabla 1.27: Representación de un nodo necesaria para recrear el mapa de red, contenido del mensaje de señalización

1.3.3.2. Bluetooth

Código invulcrado

BluetoothController: connected(), setState()
Service: mHandlerBt, sendInfoNewNodeOnNetwork()
Database: newDeviceEventNormal(), modifyNode(), updateInfoDeviceCoordinator()

Como hemos explicado antes, es ahora el coordinador quien tiene que dar el primer paso, de hecho cuando se congifura la interfaz, está incluida una ronda de conexión a los nodos inactivos. Al producirse el canal de comunicaciones tanto el nodo coordinador como el nodo normal informan a sus respectivos servicios que se ha producido un evento de nodo nuevo `EVENT_NEW_DEVICE`.

Normal Al recibir el evento conexión al coordinador vemos si éste está en la base de datos, si no lo estuviese, incluiríamos su dirección MAC para obtener su identificador y pospondríamos el anuncio del nuevo nodo hacia las aplicaciones hasta que no obtengamos el mensaje de señalización dónde se nos informa de las características del resto de nodos en la red. Si por el contrario ya habíamos cooperado antes con este nodo, anunciaremos de inmediato hacia las aplicaciones el evento de conexión al nodo coordinador. Notar que los parámetros del nodo coordinador (nombre y papel) pueden ser valores que no se correspondan con la situación actual, no es una cosa que nos deba preocupar pues en primer lugar son parámetros cuyo cambio suele ser puntual y en segundo lugar será solventado momentos después al recibir la señalización con información sobre la población en la red.

Coordinador El flujo también comienza preguntándonos si habíamos cooperado ya con el nodo al que acabamos de conectarnos. Si no estuviese en la base de datos, le

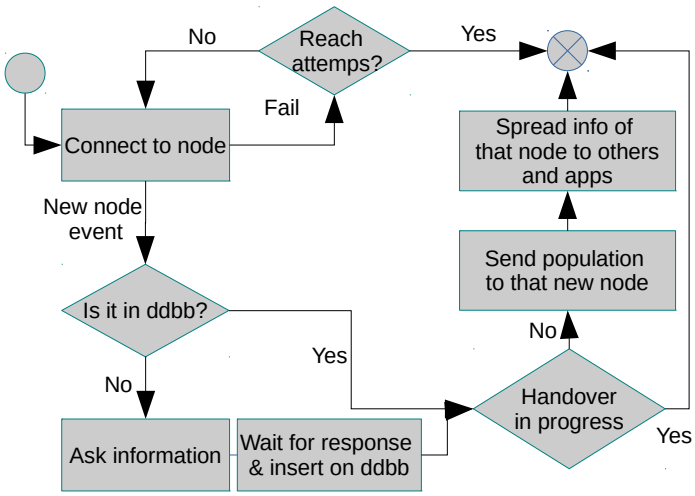

Figura 1.5: Flujograma de los procesos desencadenados en el coordinador al introducir un nodo en *Bluetooth*

enviamos un mensaje de petición de información para que nos informe de su nombre y papel, ya que su dirección MAC y su tipo (normal) lo sabemos porque o nos lo facilita el controlador o porque es deducible por el contexto. El nodo normal al recibir este mensaje envía sus datos, que nosotros procesamos como si fuese una actualización de parámetros en la red más, lo que conlleva modificar los datos del nodo que nos lo envía y la propagación de estos al resto de nodos. Además en este caso como el nodo no estaba en la red, se le envía información sobre la red cerrando el proceso de registro y se anuncia hacia las aplicaciones la entrada de un nuevo nodo, siguiendo la estructura de mensaje visto en la tabla 1.13

1.3.4. Salida de un nodo de la red

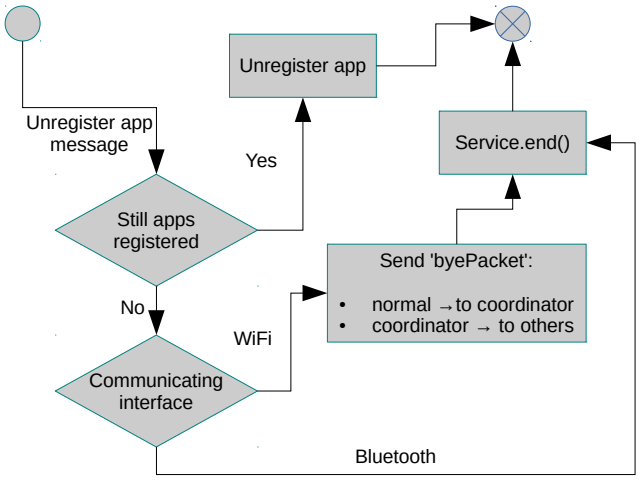

Figura 1.6: Flujo salida red

La dificultad en el proceso de salida de un nodo de la red viene marcada tanto

por la interfaz como por el tipo de nodo que se trate. Es un proceso que tiene dos lados, el nodo que se desregistra en la red y los nodos que se quedan. Un nodo se desregistra de la red cuando no quedan aplicaciones montadas sobre el servicio. Para que una aplicación se desregistre del servicio tiene que mandar una petición de *unbind* precedida de un mensaje del tipo *unregister client* (ver tabla 1.17). Con el mensaje el servicio sabrá que tiene que borrar de su base de clientes a la aplicación y la petición liberará la conexión creada por *Android* entre ambos.

El flujo que relaciona aplicación y servicio destruye a éste cuando todas las peticiones de *bind* son correspondidas con sus *unbind*, parecido pasa con las peticiones de *start* y *stop* (en este caso una única petición de *stop()* destruiría el servicio). *Android* tiene este tándem para decidir si un servicio debe seguir vivo o no, como las peticiones de *start* son disparadas por el propio servicio para realizar de forma periódica su tarea cognitiva, ha sido necesario modificar el ciclo de vida para que cuando el servicio deje de tener aplicaciones sobre él, se destruya a si mismo y no automáticamente como normalmente ocurriría.

Cuando ocurre esto, los pasos a seguir son la preparación de la base de datos para una nueva instancia del servicio. Estas acciones comprenden resetear las direcciones IP y el identificador del coordinador para cada nodo, así como el marcaje de éstos como inactivos. Por último una indicación de que el servicio ha sido parado de forma voluntaria para darnos cuenta de posibles cierres forzados, de tal forma que al instanciar de nuevo el servicio siempre tengamos una situación de partida inicial. Estas acciones no son ejecutadas si la salida es forzosa: debido a un error no manejado que provoca el cierre o debido a la falta de memoría que provoca que el gestor de *Android* cierre la aplicación. En este caso como no se ha puesto la marca de cierre voluntario del servicio, al arrancar el servicio de nuevo podemos realizar estas acciones al comienzo para desembocar en la misma situación inicial.

Entre las acciones a ejecutar por el nodo que sale de la red tenemos que dependiendo de la interfaz la forma de avisar al resto de nodos que se quedan varia:

1.3.4.1. WiFi

Código invulcrado
WifiController: sendbyePacket(), sendbyePacketReliable(), SendByePacket (AsyncTask)
Service: mHandlerWF, eventLostActionsNormal(), onDestroy(), onCreate()
Database: lostNode()

En esta interfaz tenemos una conexión intermitente, que se despliega y se retrae en los momentos en los que hay comunicación. En esta situación ante eventualidades no tenemos el mismo tiempo de reacción como lo podemos tener en *Bluetooth*. En el caso de salida voluntaria necesitamos un mecanismo que informe que un nodo va a abandonar la red.

Este mecanismo consiste en el envío de un paquete llamado *"ByePacket"* que contiene la dirección MAC del nodo que abandona la red. El envío de este paquete puede ser sobre TCP o UDP, en el caso de que el envío sea desde el nodo coordinador, sólo podremos mandarlo sobre TCP ya que los nodos normales no escuchan sobre UDP como hemos visto en 1.3.2

Las acciones a ejecutar por los nodos que se quedan en la red son:

Estructura mensaje desregistro en la red *WiFi*

Tipo de paquete Valor "BYE" del enumerado "packetType".

Bye packet Un contenedor que contiene al mensaje del tipo que se indicó en el campo "Tipo de paquete":

Dirección MAC Una *cadena de caracteres* cuyo valor es la dirección MAC de *Bluetooth*, clave que identifica al nodo de manera unívoca.

Tabla 1.28: Campos del paquete "ByePacket"

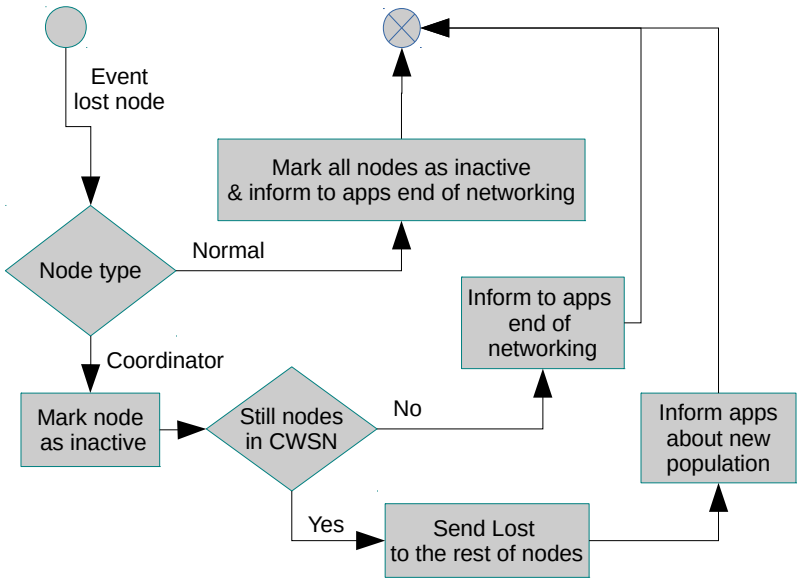

Figura 1.7: Flujo evento pérdida de nodo manejado disparado por evento del controlador desde los nodos que se quedan en la red

Normal Al recibir de manera fiable el paquete *"ByePacket"* por parte del coordinador, el controlador generará un evento de pérdida de nodo que llegará al servicio donde procederemos a marcar a todos los nodos como inactivos y demás información volátil de la CWSN, por otra parte informamos a las aplicaciones registradas de la pérdida del nodo y de la salida de la red. Si recibimos mediante señalización la pérdida de un nodo, procedemos a marcarlo como inactivo y a informar a las aplicaciones registradas del nuevo mapa de red.

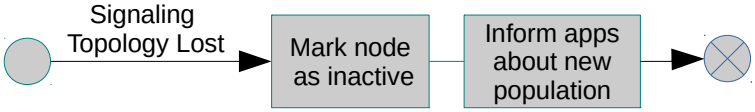

Figura 1.8: Flujo evento pérdida de nodo disparado por señalización manejado desde un nodo normal

Coordinador Al recibir el paquete *"ByePacket"* se generará el evento de pérdida de nodo que se entregará al servicio. Marcaremos este nodo como inactivo y veremos si aún quedan nodos en la red, si la respuesta es afirmativa enviaremos un mensaje de señalización a cada nodo restante en la red informando de la pérdida, así ellos podrán recomponer el mapa de red e informar a sus aplicaciones, informamos también a nuestras aplicaciones registradas del evento y de la nueva lista de nodos. Si la respuesta es negativa, nos limitamos a informar a las aplicaciones registradas del evento de pérdida y del evento de salida de la red.

1.3.4.2. Bluetooth

Código invulcrado

BluetoothController: connectionLost(), setState()
Service: mHandlerBt, eventLostActionsNormal(), onDestroy(), onCreate()
Database: lostNode()

En este caso tenemos una conexión abierta en todo momento, por lo que ante cualquier eventualidad nos daremos cuenta inmediatamente de que se ha perdido la conexión, lo que hace innecesario el mecanismo descrito en la sección anterior. Tanto si se trata de una salida voluntaria como involuntaria, se destruye tanto la aplicación como el servicio, se liberan todos los recursos. Al cerrarse el controlador de *Bluetooth*, los canales de comunicación abiertos desaparecen lo que acarrea que el otro extremo de la comunicación se de cuenta de la desaparición de éste y se lance el evento de pérdida de nodo desde ambos extremos.

Las acciones a ejecutar por los nodos que se quedan en la red son:

Normal Si somos un nodo normal, el evento elevado hasta el servicio por el controlador significa que hemos perdido conexión con nuestro coordinador (que acaba de abandonar la red) y por tanto con toda la red, por lo tanto marcamos a todos los nodos como inactivos borrando además todos los datos relativos a la CWSN que acabamos de abandonar. Informamos a las aplicaciones registradas que hemos dejado de formar parte de la CWSN. Por otro lado, al recibir la señalización que indica la pérdida de un nodo seguimos el flujo visto en la figura 1.8

Coordinador Si por el contrario somos un nodo coordinador, significa que hemos perdido la conexión con un nodo normal con el que teníamos comunicacion directa y debemos informar al resto de nodos de esta eventualidad enviando un mensaje de señalización con la dirección MAC del nodo saliente para que ellos puedan rehacer el mapa de red. Marcamos también al nodo del que acabamos de perder la comunicación como inactivo e informamos a las aplicaciones registradas del evento y la nueva lista de nodos. Si todos los nodos de nuestra base de datos están marcados como inactivos, significa que acabamos de perder la comunicación con el último nodo y por tanto dejamos de estar en CWSN situación que informamos a las aplicaciones registradas de manera análoga.

1.3.5. Actualización de parámetros en la red

Código invulcrado

Service: m2service, spreadNodeChangeOnNetwork(), updateInformationToCoordinator()

Los parámetros del servicio se pueden dividir en dos grupos, los propios del servicio y los de las aplicaciones de los cuales uno de ellos es único y por tanto está compartido entre todas ellas (ver tabla 1.29).

Parámetros exclusivos servicio	
Nombre del nodo	Una *cadena de caracteres* que represente el nombre del nodo en la red.
Tipo de nodo	Un entero cuyo valor '0' se traduce por normal y cuyo valor '1' corresponde a coordinador.
Periodo tarea cognitiva	Un *double* que expresa el número de segundos que transcurren entre ejecuciones de la tarea cognitiva.
Parámetros aplicación	
Papel del nodo	Un entero para representar el papel que juega el nodo en la red, '0': usuario secundario mientras que el valor '1' representa al usuario primario.
Código de aplicación	Una *cadena de caracteres* para discernir entre mensajes enviados por distinas aplicaciones.

Tabla 1.29: Parámetros del servicio

Los primeros parámetros son ajustes del servicio que traspasan a cualquier aplicación, sin embargo los segundos si son propios de cada aplicación. El primero expresa las necesidades de comunicación y el segundo nos sirve para filtrar mensajes en el servicio y entregar el contenido que está dirigido a esa aplicación en concreto. El parámetro "papel del nodo" en realidad es un parámetro compartido, es decir si varias aplicaciones están registradas en el servicio cada una habrá aportado un valor distinto, como este valor es único, elegimos el que es más adecuado a las necesidades de comunicación de la aplicación más restrictiva. En otras palabras si una aplicación pide ser "usuario primario" y otra aplicación pide ser "usuario secundario" el valor del parámetro será "usuario/nodo primario".

La primera forma de actualizar los parámetros de aplicación es con el registro de la aplicación en el servicio (ver sección 2.2.1) o en cualquier momento enviando un mensaje desde la aplicación al servicio del tipo `REGISTER_EXCHANGE` (visto en la tabla 1.5) que desembocará en una contestación con todos los parámetros del servicio. Con respecto a los parámetros exclusivos del servicio, la forma de actualizarlos está un poco escondida en la API, ya que son ajustes propios del servicio y no de la aplicación y por tanto sometidos a cambios poco frecuentes. Normalmente estos parámetros están guardados en base de datos y se recuperan al recibir un mensaje del tipo `REGISTER_EXCHANGE`, sin embargo si en este mensaje viene informado el campo *obj* con un valor booleano a '*true*' entonces en vez de coger los parámetros de la base de datos, los extraemos del mensaje. El detalle del mensaje ya se definió en la tabla 1.19.

La actualización de estos parámetros (tanto cuando sólo actualizamos los de aplicación o todos) viene condicionada a si estamos cooperando ya en una red o no. Si no estamos en ninguna red los valores pasados sobreescribiran los existentes. En cambio si estamos en una situación de red, los parámetros "papel del nodo" y "periodo tarea cognitiva" pueden no ser actualizados, su actualización depende por tanto del valor actual, si es más restrictivo se podrán actualizar. Es decir para el "papel del nodo" sólo actualizaremos si el cambio pedido es a "nodo primario" y para "periodo tarea cognitiva" sólo actualizaremos si el periodo requerido es más pequeño que el actual. Con respecto al parámetro "tipo de nodo" una vez se está en una CWSN no se permiten cambios ya que dejaríamos a la red sin coordinador o añadiriamos otro coordinador, situaciones que no son de interés. Lo anterior no aplica, salvo el cambio de tipo de nodo, en la situación en que sólo hay una aplicación registrada, situación en la que actualizaremos los parámetros al no haber conflicto.

Con respecto a la actualización de estos parámetros al resto de nodos, una vez validados y actualizados según las reglas descritas, si son de interés general ("nombre del nodo", "papel del nodo") son enviados al coordinador para que este los distribuya como una actualización más del mapa de red.

En *Bluetooth* puede darse la situación de que si somos un nodo normal los demás nodos nos vean con parámetros antiguos, esto es debido a que si actualizamos nuestros parámetros fuera de una CWSN, estos obviamente no son propagados a la red, al ser incluidos en la red por el coordinador, que ya nos conoce de veces anteriores, no tiene manera de saber que hemos actualizado los parámetros. Como impera la minimización del número de mensajes pasados por la red, éste no nos preguntará, sino que sacará una copia de los parámetros de su base de datos, que serán los mismos que tendrá el resto de la red, pues son la copia desactualizada y distribuida del coordinador. Esta situación es muy poco frecuente por lo que teniendo en mente la minimización del tamaño de mensajes y el número de éstos, se ha preferido no implementar mecanismos que hagan mantener actualizados en todo momento los parámetros del nodo ("nombre del nodo", "papel del nodo" y "tipo de nodo" aunque este último es deducible por el contexto) en favor de una mayor eficiencia de red. Esta situación no ocurre en *WiFi* ya que la forma de registro en la red conlleva el envío de un paquete para registrarse en el que enviamos toda la información del nodo, por lo que sobre *WiFi* se puede actualizar en cualquier momento y sobre *Bluetooth* sólo en momentos de cooperación en red. No obstante la situación de desactualización se soluciona cuando hay un cambio de contexto con interfaz destino *WiFi*.

1.3.6. Intercambio de mensajes

La transmisión y recepción de datos se realizan a través del intercambio de mensajes definidos en la API. Podemos destacar tres acciones: el envío, la recepción y una mezcla de ambos reservada sólo al coordinador que es el re-envío o *forward* de mensajes.

1.3.6.1. Recepción de mensajes

Código invulcrado

Service: m2service, mHandler<controlador>, incomingContentNormal(), incomingDataMessageToApp()

Empezamos explicando este proceso por ser el menos complejo y nos que nos ayuda a introducir algunas de las piezas que veremos en el proceso de envío. Al recibir datos se genera el evento `INTERFACE_STATE_RECEIVING` que es manejado en el servicio, para que este informe a las aplicaciones. Cuando ha terminado de recibir todo el mensaje, el controlador proporciona nuevamente al servicio esta información que dependiendo del controlador, incluirá `EXTRAS` distintos (ver tabla 1.30).

Campos mensaje desde un controlador (*Bluetooth* o *WiFi*) al servicio

What	*arg1*	*arg2*	*obj*	*replyTo*
2 = MESSAGE_READ	Número de *bytes* leídos	No usado/indiferente	*bytes* leídos de la red	No usado/indiferente

Extras, controlador: *WiFi*

key	*value*
device_ip_address	Una *cadena de caracteres* con la direción IP del nodo que nos envía los datos

Extras, controlador: *Bluetooth*

key	*value*
device_name	Una *cadena de caracteres* con el nombre proporcionado por los servicios de *Bluetooth* de *Android* (no es relevante)
device_address	Una *cadena de caracteres* con la dirección MAC de *Bluetooth* del nodo que nos envía los datos

Tabla 1.30: Mensaje recibido por la red entregado por ambos controladores al servicio

Una vez tenemos los *bytes* leídos en el servicio los procesamos. Este procesamiento puede ser tan simple como entregar el contenido del mensaje a la aplicación correspondiente o reenviar el mensaje al siguiente eslabón de la cadena (sección explicada en la página 69). El tratamiento del mensaje depende del tipo de nodo. Sin embargo, la presentación del mensaje hacia la aplicación si éste está dirigido hacia nosotros es común. Para ello necesitamos obtener el "Código de aplicación" (ver tabla 1.31) que nos permite recuperar el *Messenger* de la aplicación. Así filtramos y entregamos a cada aplicación los mensajes que le interesan. También, gracias al campo "Desde" sabemos quién es el originador del mensaje. Una vez hemos recuperado la aplicación destinataria y el nodo que envío el mensaje, podemos trasladar esta información a la aplicación correspondiente (ver tabla 1.10).

1.3.6.2. Envío de mensajes

Código invulcrado

Service: m2service, builtDataMessage(), getNextNodes(), send(), mHandler<controlador>, outgoingContentProgress()
DataMessageQueue: add(), hashMessage(), changeStatus(), remove()

Para enviar datos a través del servicio cognitivo, la aplicación debe enviar un mensaje a través de la API con la siguiente estructura (ver tabla 1.8).

Una vez entregado el mensaje al servicio se empieza a procesar. Asumimos que el mensaje que se quiere enviar es de un contenido que se puede codificar como una *cadena de caracteres*, por tanto éste es el tipo de dato que se espera en el campo *obj*

del mensaje (*payload*), si no el servicio devolverá un mensaje de error a la aplicación explicando que el contenido no puede manejarlo. En segundo lugar se comprueba que todos los valores de la lista son nodos de la red y están activos, también comprobamos que estamos en una CWSN. Si no se superase alguna de estas otras validaciones se entregaría al igual que antes un mensaje de error a la aplicación con el código y descripción pertinentes abortándose el proceso en curso. Si no hay ningun problema, confeccionamos el mensaje que se va a enviar a la red cuyo formato vemos en la tabla 1.31

Estructura mensaje datos

Tipo Valor "DATA" del enumerado "MessageType".

Desde Una *cadena de caracteres* con la dirección MAC de *Bluetooth* del nodo.

Lista destinatarios Una lista de *cadena de caracteres* con la lista con las direcciones MAC de *Bluetooth* de los nodos destinatarios.

Código de aplicación Una *cadena de caracteres* con el código de aplicación para filtrar el mensaje en destino.

Marca de tiempo Un *long* con la representación del momento en el cual la aplicación entregó el mensaje al servicio.

Mensaje de datos Un contenedor que contiene al mensaje del tipo que se indicó en el campo "Tipo":

Payload Una *cadena de caracteres* con el contenido del mensaje.

Tabla 1.31: Campos del mensaje de datos

Una vez tenemos el mensaje lo guardamos en una cola, lo que nos ayuda a calcular el tiempo de llegada de los mensajes de datos por parte de las aplicaciones y es necesario para ofrecer a la aplicación una confirmación positiva o negativa sobre la entrega a los nodos destinararios para cada mensaje. Como esta confirmación puede no ser inmediata, necesitamos almacenar de alguna manera el mensaje para poder recuperarlo cuando nos llegue ésta. Para guardarlo y rescatarlo cuando llegue el momento nos ayudamos de una clave, un *hash* calculado en base a la marca de tiempo, el contenido del mensaje y la dirección MAC de *Bluetooth* del nodo que envía el mensaje.

El siguiente punto es ver si podemos enviar el mensaje por la red, para ello debemos estar registrados en una CWSN y no estar en una situación de cambio de contexto, si se da esta situación marcamos el mensaje como "pendiente de envío" en la cola. Si podemos enviarlo lo marcamos como "en envío" y se procede al envío. Para ello necesitamos saber quién o quienes son los siguientes nodos en la cadena de envío. Aunque gracias a que cada nodo tiene información sobre cómo alcanzar a otros nodos, nuestra red está centralizada en la figura del coordinador, por lo que la comunicación entre nodos no es posible siendo precisa una comunicación de saltos en los que el coordinador juega el papel central, siguendo la recomendación vista en CB ([1]) para incrementar la eficiencia. No obstante, hay casos en que esta comunicación de saltos viene forzada por la interfaz de comunicación de uso, estamos hablando de *Bluetooth* cuya única tipología posible es la de estrella (esclavo/maestro). El encaminamiento

por tanto depende del tipo de nodo:

Normal El siguiente eslabón de la cadena siempre es nuestro nodo coordinador, tanto si el mensaje es dirigido a él, como si lo es a otros nodos.

Coordinador El coordinador tiene comunicación directa con todos los nodos, en *WiFi* ocurre siempre y en *Bluetooth* tenemos la restricción de hasta siete nodos, si no introducimos la figura del "coordinador intermedio", nodo que se comporta como concentrador de mensajes en su *piconet* y que a su vez está en otra *piconet* donde se encuentra el coordinador. En estos momentos como sólo hay un salto, tenemos contacto directo con cualquier nodo no importando la interfaz en la que nos encontremos. Por tanto los siguientes eslabones de la cadena de entrega son los mismos nodos que están en el campo "Lista Destinatarios" del mensaje a enviar.

Una vez tenemos éste y a quién se lo vamos a enviar, procedemos al envío (en la siguente sección veremos que necesitamos un parámetro más pero que por el momento podemos obviar). Enviar es basicamente pasar como parámetros el mensaje al método *write()* del controlador traduciendo los nodos a la dirección de comunicación de la interfaz, es decir si tenemos que enviar un mensaje a una serie de nodos de los cuales tenemos sus identificadores, tendremos que traducir estos identificadores a las direcciones IP en el caso de *WiFi* o a las direcciones MAC en *Bluetooth*. Una vez que el mensaje ha sido escrito en la red por el controlador, este mismo informa al servicio a través de un mensaje del tipo `MESSAGE_WRITE` con resultado del envío, enviándonos el propio mensaje de vuelta más una lista de las direcciones de comunicación a las cuales no ha podido entregar el mensaje (ver tabla 1.32).

Campos mensaje desde un controlador (*Bluetooth* o *WiFi*) al servicio

What	*arg1*	*arg2*	*obj*	*replyTo*
3 = MESSAGE_WRITE	número mensajes escritos OK	número mensajes escritos NOK	*bytes* enviados a través de la red	No usado/indiferente

Extras

key	*value*
requester	Una *cadena de caracteres* que representa el *messenger* de la aplicación que origina el mensaje
addressNotDelivered	Una lista de *cadena de caracteres* con las direcciones MAC de *Bluetooth* o direcciones IP de los nodos a los cuales el envío del mensaje ha resultado fallido, según el controlador que nos envíe el mensaje

Tabla 1.32: Mensaje resultado envío de datos desde el controlador al servicio

Dependiendo de quienes sean los destinatarios del mensaje, el mensaje habrá llegado a su destino (no hay saltos de por medio) y por tanto no necesitamos más confirmación de entrega que la que ya nos da TCP de por si. Si no tuviesesemos contacto directo con los nodos a los que queremos enviar el mensaje necesitamos esperar a que nos confirmen el resultado de la entrega de éste.

No es necesario recibir mensaje ACK En este primer caso, al ser una entrega directa, si somos un nodo normal el mensaje tiene como único destinatario nuestro coordinador. Si somos un nodo coordinador el mensaje ha sido entregado en mano a todos los nodos. Como ya se ha terminado el flujo de envío del mensaje, procedemos a borrarlo de nuestra cola de mensajes gracias al *hash* y a entregar un mensaje a la aplicación con información sobre la entrega (ver tabla 1.9).

Es necesario mensaje ACK En el segundo caso, el mensaje envíado requiere de un salto en la red, requiere de un reenvío. La casuística de cómo se producen este tipo de mensajes será explicada en la siguiente sección, en ésta nos limitamos a ver el proceso de recepción del ACK (*Acknowledgment*) vista desde un nodo normal. En este caso al recibir por parte del controlador el mensaje (ver tabla 1.32) procedemos a su marcaje en la cola como "pendiente de ack" quedando a la espera de recibir la confirmación del envío del siguiente salto. Este mensaje de señalización tiene la composición descrita en la tabla 1.33

Estructura mensaje ACK

Tipo Valor "RESPONSE" del enumerado "MessageType".

Mensaje de respuesta Un contenedor que contiene al mensaje del tipo que se indicó en el campo "Tipo":

- **What** Valor "ACK" del enumerado "AskType".
- **Mensaje de ACK** Un contenedor que contiene al mensaje del tipo que se indicó en el campo "What":
  - **Hash** Un *long* con el identificador único de mensaje.
  - **Número mensajes escritos OK** Un entero con el número de mensajes que han sido correctamente entregados.
  - **Lista de direcciones MAC NOK** Una lista de *cadena de caracteres* con las direcciones MAC de *Bluetooth* de los nodos a los cuales no se ha podido entregar el mensaje.

Tabla 1.33: Campos del mensaje ACK

Al recibir la confirmación procedemos a borrar de la cola el mensaje de datos utilizando el *hash* leído de la señalización y a comunicar a la aplicación el resultado del proceso de envío. (El detalle puede verse en la tabla 1.9)

1.3.6.3. Forward de mensajes

Código invulcrado

Service: mHandler<controlador>, incomingContentCoordinator(), outgoingContentProgressCoordinator()

Este proceso es una mezcla de una recepción y un envío de mensajes, reservada al papel de coordinador. Este proceso comienza con la recepción (en el nodo coordinador) de un mensaje originario en un nodo normal cuya lista de nodos destinatarios atañe a más nodos de los dos implicados en esta comunicación. En esta lista puede estar o no

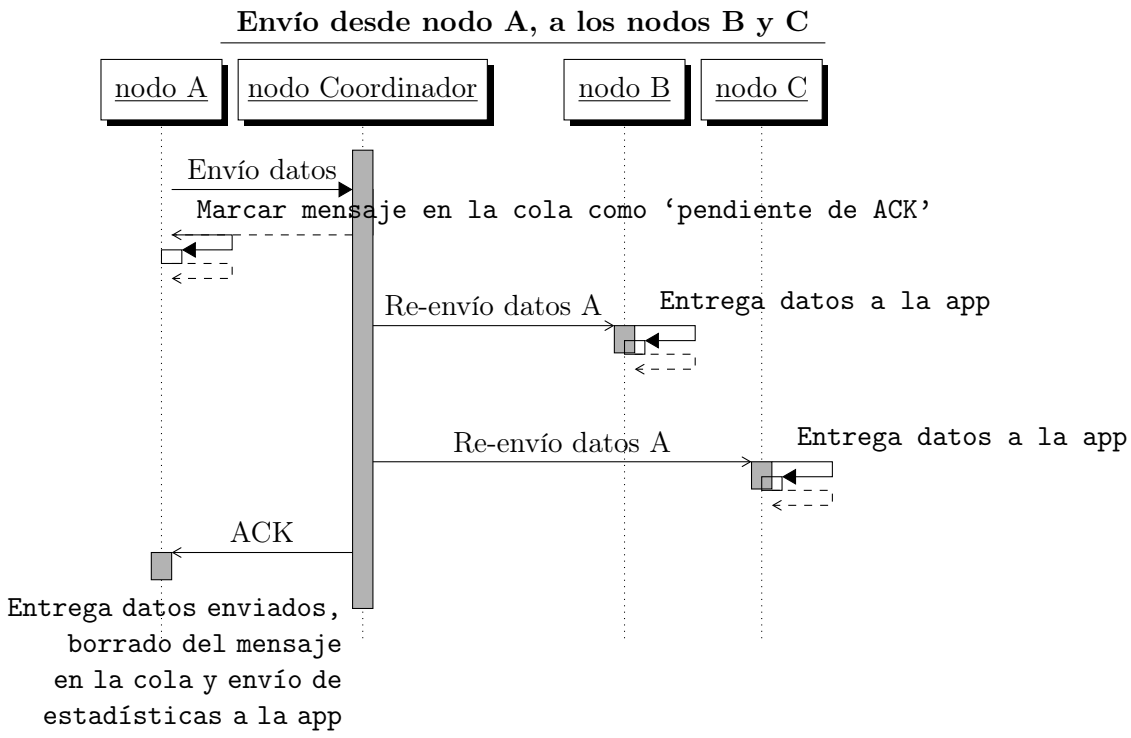

Figura 1.9: Ejemplo envío con necesidad de ACK

incluido el coordinador, si está incluido como paso inicial presentaremos el mensaje a la aplicación como hemos descrito en la recepción de mensajes. Acto seguido procederemos al re-envío de éste, para ello como vimos en la sección anterior tenemos que conocer cuáles son los siguientes nodos a los que enviar el mensaje (prestando especial atención a que si es un mensaje dirigido a todos, no volver a enviarlo al nodo originario de la comunicación), el mensaje a enviar que es el mismo que acabamos de recibir (sin hacerle ningún cambio) y por último el parámetro que antes obviamos.

Esta elusión tiene como motivación haber facilitado la lectura de secciones anteriores al no añadir otra pieza más al rompecabezas. Nuestro flujo de trabajo necesita transmitir a través de todas las capas (servicio, controlador y capas inferiores, si intervienen) qué aplicación es la originaria del mensaje para poder ofrecerle una respuesta cada vez que envie datos a través del servicio. En el caso de un re-envío no hay aplicación originaria del mensaje en el nodo que acaba de recibir un mensaje que tiene que retransmitir, por ello no se puede utilizar el "Código de aplicación" grabado en el mensaje de datos, además su uso implica que no podría haber dos aplicaciones registradas a la vez con el mismo código. El parámetro por tanto es una simple *cadena de caracteres* que representa, o al *messenger* de la aplicación o si ésta no existe, una *cadena de caracteres* especial que marque la situación de reenvío.
De esta manera uniformamos el valor de este campo al tratarse en ambos casos de *cadenas de caracteres* e indirectamente evitamos serializar un objeto complejo como es el *Messenger* de *Android* o ubicarlo en un sitio que no está pensado para él. Al intervenir en este proceso varias capas de la arquitectura si hubicasemos el objeto en el campo (*msg.replyTo*) se usaría con otro propósito al pensado por la plataforma *Android*, lo que hace poco claro el código.
Es necesario transmitir este parámetro a través de todo el flujo de envío ya que son

procesos que pueden no resolverse en la misma petición, además son sin memoria y pueden ser disparados por distintas aplicaciones en cualquier momento donde además el preservamiento del orden entre distintos procesos de envío no está garantizado.

Las *cadenas de caracteres* especiales que representan la situación de reenvío y que son claves para que el flujo del proceso actue en conseguencia son: `‘‘forward-me’’` y `‘‘forward-not-me’’` para referirse tanto a la situación en la que el coordinador forma parte de la lista del mensaje que se reenvía, tanto a cuando no.

Al igual que en un envío normal, almacenamos el mensaje en la cola y lo enviamos. Al recibir por parte del controlador el mensaje descrito en la tabla 1.32 en vez de trasladar el mensaje a la aplicación (dado que no existe) como se hace en el envío, es aquí dónde confeccionamos el mensaje de confirmación (ver tabla 1.33) que enviaremos al nodo originario del mensaje de datos.

1.3.7. Sensado del entorno compartido y política cognitiva

Los parámetros del entorno que son de nuestro interés son el RSSI (*Received Signal Strength Indication*) (intensidad de señal) de la red *WiFi* a la cual estamos conectados y la tasa de envío de mensajes de datos. El sensado del entorno se realiza en varios puntos de la arquitectura y son compartidos mediante señalización (ver tabla 1.34) con el coordinador de forma periódica. Éste al recibirlos dependiendo de la prioridad con la que vengan marcados se limita a guardarlos en su base de datos o a hacer un análisis rápido de la situación que desemboque en la toma de ciertas decisiones.

Estructura mensaje SCAN

Tipo Valor “RESPONSE” del enumerado “MessageType”.

Mensaje de respuesta Un contenedor que contiene al mensaje del tipo que se indicó en el campo “Tipo”:

- **What** Valor “SCAN” del enumerado “AskType”.
- **Mensaje de SCAN** Un contenedor que contiene al mensaje del tipo que se indicó en el campo “What”:
  - **Prioridad** Un enumerado (Priority) con la urgencia del sensado, toma dos valores Priority.TASK para sensados normales y Priority.URGENT para marcar que la información del entorno es sensible.
  - **RSSI** Un entero con el nivel de señal de la red *WiFi* a la que estamos conectados (-9999 si no estamos en ninguna).
  - **Intervalo promedio envío de mensajes** Un *double* con el resultado promedio móvil del intervalo medido en segundos entre la llegada de mensajes de datos por parte de las aplicaciones.

Tabla 1.34: Campos del mensaje SCAN

1.3.7.1. Parámetro del entorno: RSSI

Código invulcrado

Service: normalCognitiveTask(), mWifiAdapter.getConnectionInfo()

Para medir la intensidad de señal podemos utilizar de manera activa la API de *Android* obteniendo el valor de la fuerza de la señal medida en dBm. También de una manera indirecta, aprovechamos los mensajes que lanza el sistema *Android* con información acerca de este parámetro. El registro se efectúa cuando se inicializa el servicio y nos desregistramos (para no dejar *memory leaks*) al finalizar el servicio. Así las diferentes políticas cognitivas tienen disponible este parámetro a lo largo de toda la vida del servicio.

En nuestra arquitectura utlizamos estos dos puntos de medida para este parámetro, el primero está localizado en la tarea cognitiva dónde lo guardamos en la base de datos para usos futuros. El otro punto reside en las notificaciones, éstas nos ayudan a actuar con el menor tiempo de respuesta, ya que si detectamos una situación que entrañe riesgo podemos compartir esta información sin esperar a la tarea cognitiva donde realizamos entre otras cosas el sensado del entorno.

1.3.7.2. Parámetro del entorno: Intervalo promedio envío de mensajes

Código invulcrado

Service: normalCognitiveTask()
DataMessageQueue: getAvgArrivalRateUpdateTillNow()

En este caso la medición se va confeccionado cada vez que una aplicación entrega un mensaje de datos al servicio para su envío, al añadirlo a la cola registramos el tiempo que ha pasado desde la última vez, si tenemos suficientes registros olvidamos el último antes de añadir este nuevo, esto nos permite realizar promedios móviles que se centren en una situación cercana al momento actual para extraer información reciente y no condicionada excesivamente por eventos pasados.

En la tarea cognitiva (que recordamos que es periódica) calculamos la media de estos valores. Se da la situación de que el promedio sólo cambia al incluir nuevos mensajes en la cola, desembocando en que si no se producen nuevas inclusiones se obtiene el mismo valor del intervalo de envío de mensajes en ejecuciones consecutivas de la tarea cognitiva, desvirtuando este valor. Necesitamos algún mecanismo que sin desfigurar los datos, refleje esta situación de no envío y por tanto evidenciar un aumento del promedio.

Para ello en el momento de la transmisión de los parámetros del entorno al coordinador, obtenemos el promedio y registramos de nuevo el tiempo así podemos recalculamos el intervalo tal y como si hicieramos un falso envío pero sin incluir a éste en los datos para no desfigurar el resultado en sucesivos envíos (que se hayan producido realmente).

1.3.7.3. Política cognitiva

Código invulcrado

Service: mayTakeAdvantageOfBT() mayTakeAdvantageOfWiFi(), sendTroublingRssi(), coordinatorCognitiveTask()

La política cognitiva implementada es competencia del nodo coordinador. Esta política tiene en cuenta tanto los parámetros propios del nodo coordinador como el sensado del entorno de los demás nodos. El análisis de los datos de sensado recae fundamentalmente

en el nodo coordinador, sin embargo para evitar situaciones de pérdida de nodos, el nodo normal avisa de forma activa si su RSSI decae por debajo de un umbral.
El cambio de interfaz obedece a estos motivos:

De WiFi a Bluetooth Por pérdida de la señal.

De Bluetooth a WiFi Por el aumento de la tasa de envío de mensajes de datos en la red.

1.3.8. Cambios de contexto

Con la información recogida del entorno tanto por el mismo nodo como por sus compañeros podemos tomar decisiones sobre la estabilidad de la red y conforme a experiencias pasadas colocar a ésta en el lugar que mejor se adapte. Necesitamos una serie de protocolos que mantengan la integridad de la red y marquen los pasos a seguir cuando hay un cambio de interfaz.

1.3.8.1. Cambio de contexto con interfaz destino WiFi

Código invulcrado

Service: coordinatorCognitiveTask(), mayTakeAdvantageOfWiFi(), contextChangeToWiFi(), SetupWifiInterface (asyncTask), mHandlerWF, scheduleTaskWifi(), sendEndSwitchMessage(), endContextualChange()

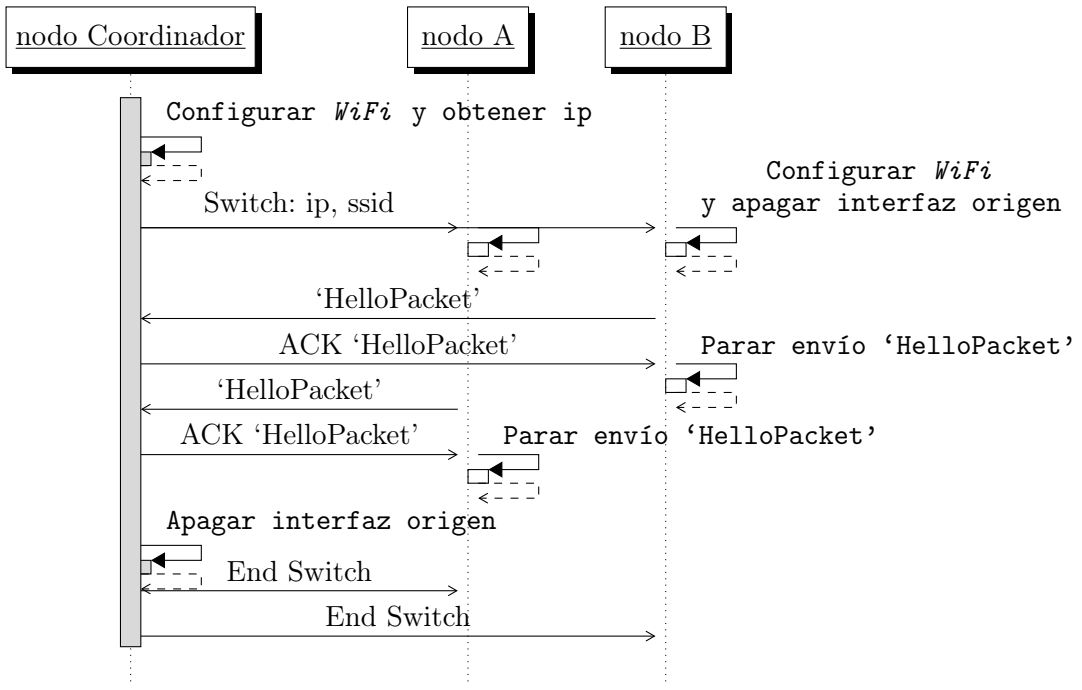

Figura 1.10: Ejemplo cambio de contexto hacia *WiFi*

Cuando el coordinador ejecuta su tarea cognitiva va rescatando de su base de datos para cada nodo los últimos escaneos de éste obteniendo un promedio y una tendencia de los datos de sensado. Evalúa estos datos y decide si es necesario un cambio de interfaz. En caso positivo, obtenemos una lista de los nodos activos en la red, los

marcamos como inactivos e informamos a las aplciaciones sobre el nuevo evento (ver mensaje en la API) y configuramos la interfaz de *WiFi*. Para ello utilizamos el mismo prodecimiento usado en 1.3.2 con unos pasos extras: una vez nos hemos conectado a la red enviamos un mensaje de señalización (a través de la interfaz actual, aún *WiFi* no es efectiva) a la lista de nodos que sacamos informandoles del cambio y de la información necesaria para llevarla a cabo, (ver tabla 1.35)

Estructura mensaje SWITCH hacia *WiFi*

Tipo Valor "SIGNALING" del enumerado "MessageType".

Desde Una *cadena de caracteres* con la direción MAC de *Bluetooth* del nodo.

Mensaje de señalizacion Un contenedor que contiene al mensaje del tipo que se indicó en el campo "Tipo":

- **Señal** Valor "SWITCH" del enumerado "SignalType".
- **Mensaje de SWITCH** Un contenedor que contiene al mensaje del tipo que se indicó en el campo "Señal":
  - **Interfaz destino** Valor "TO_WIFI" del enumerado "ToInterface".
  - **SSID** Una *cadena de caracteres* con el nombre de la red *WiFi* a la que se va a realizar el cambio.
  - **BSSID** Una *cadena de caracteres* con la dirección MAC del punto de acceso para evitar cualquier equívoco.
  - **Dirección IP coordinador** Una *cadena de caracteres* con la dirección IP del nodo.

Tabla 1.35: Campos del mensaje SWITCH hacia *WiFi*

Al terminar de configurar la interfaz (siendo ésta ya efectiva), dejamos programado un *timeout* que dispare el fin de cambio contextual para evitar bloqueos porque algún no responda. A partir de ahora esperamos a que los nodos normales configuren su interfaz de *WiFi* (antes apagarán su interfaz en curso, en este caso *Bluetooth*) y vayan entrando en la nueva red tal y como se contó en las secciones 1.3.2 y 2.2.3

El coordinador al recibir los *"HelloPacket"* en vez de seguir el flujo normal que consiste informar a todos los nodos de la nueva entrada y al nuevo nodo informarle sobre el mapa de red, nos limitamos a enviarle un mensaje de confirmación indicándole que hemos recibido el *"HelloPacket"* para que éste pare su envío, así evitamos enviar muchos mensajes de señalización de una situación que es probable que cambie. Estos paquetes están enviados directamente a la IP del coordinador (recogida del mensaje de señalización de cambio) para que haya más posibilidades de recepción. Si por algún casual el nodo normal terminase su ráfaga y no obtuviese respuesta el mismo entendería que el proceso de cambio de contexto ha acabado de manera insatisfactoria. El detalle de este mensaje de confirmación de entrada en la red en situación de cambio se ve en la tabla 1.36

Si antes de que salte el *timeout* programado, hemos recibido todos los *"HelloPacket"* de todos los nodos que estaban formando la red, procedemos a finalizar el cambio de contexto, si no esperamos a que finalice el *timeout*, situación en la que habremos perdido algún nodo.

Estructura mensaje confirmación cambio de contexto

Tipo Valor "RESPONSE" del enumerado "MessageType".

Mensaje de respuesta Un contenedor que contiene al mensaje del tipo que se indicó en el campo "Tipo":

What Valor "SWITCH" del enumerado "AskType".

Tabla 1.36: Campos del mensaje de confirmación entrada (*WiFi*) / interfaz lista (*Bluetooth*)

Coordinador Al dispararse el final del cambio de contexto, apagamos la interfaz *Bluetooth* y liberamos todos los recursos. Hemos mantenido la interfaz arriba hasta el último momento para no perder mensajes de datos si algún nodo actúa de manera incorrecta y envía estos mensajes en pleno cambio de conexto. Acto seguido se envía el mapa de red por medio de señalización a todos los nodos que formen ahora la red (idealmente todo los que la formaban antes del cambio), cuyo detalle se aprecia en la tabla 1.37

Estructura mensaje fin cambio de contexto

Tipo Valor "SIGNALING" del enumerado "MessageType".

Desde Una *cadena de caracteres* con la direción MAC de *Bluetooth* del nodo (coordinador).

Mensaje de señalizacion Un contenedor que contiene al mensaje del tipo que se indicó en el campo "Tipo":

Señal Valor "TOPOLOGY" del enumerado "SignalType".

Mensaje de mapa de red Un contenedor que contiene al mensaje del tipo que se indicó en el campo "Señal":

Cambio Valor "ALL_END_SWITCH" del enumerado "TopologyChange".

Representación de un nodo Mensaje con los cambios que sirven para representar a nodo (ver tabla 1.27). Tendremos tantas representaciones como nodos haya en la red.

Tabla 1.37: Campos del mensaje finalización cambio de contexto

Notar que este mensaje es diferente para cada nodo ya que lista de nodos vista desde cada nodo difiere en que él mismo no se encuentra en ésta pero el resto sí. Los últimos pasos para finalizar el cambio de contexto es intentar enviar los mensajes encolados (los mensajes que hayan entregado las aplicaciones mientras que el proceso de cambio de contexto esté vigente) y publicar hacia las aplicaciones registradas en el servicio el evento del fin de cambio de contexto que incluye la nueva interfaz, el estado de ésta y la lista de nodos que forman la red en este momento. Así como el vaciado de la cola de mensajes pendientes de envío.

Normal Al recibir el mensaje de señalización de cambio de contexto, procedemos a cancelar (de nuevo) el envío de *"HelloPacket"* por si no hubiesemos recibido el mensaje visto en la tabla 1.36, informar a las aplicaciones el evento de fin de cambio de contexto

con el nuevo mapa de red leído del mensaje de señalización y proceder al envío de los mensajes encolados mientras haya durado este proceso.

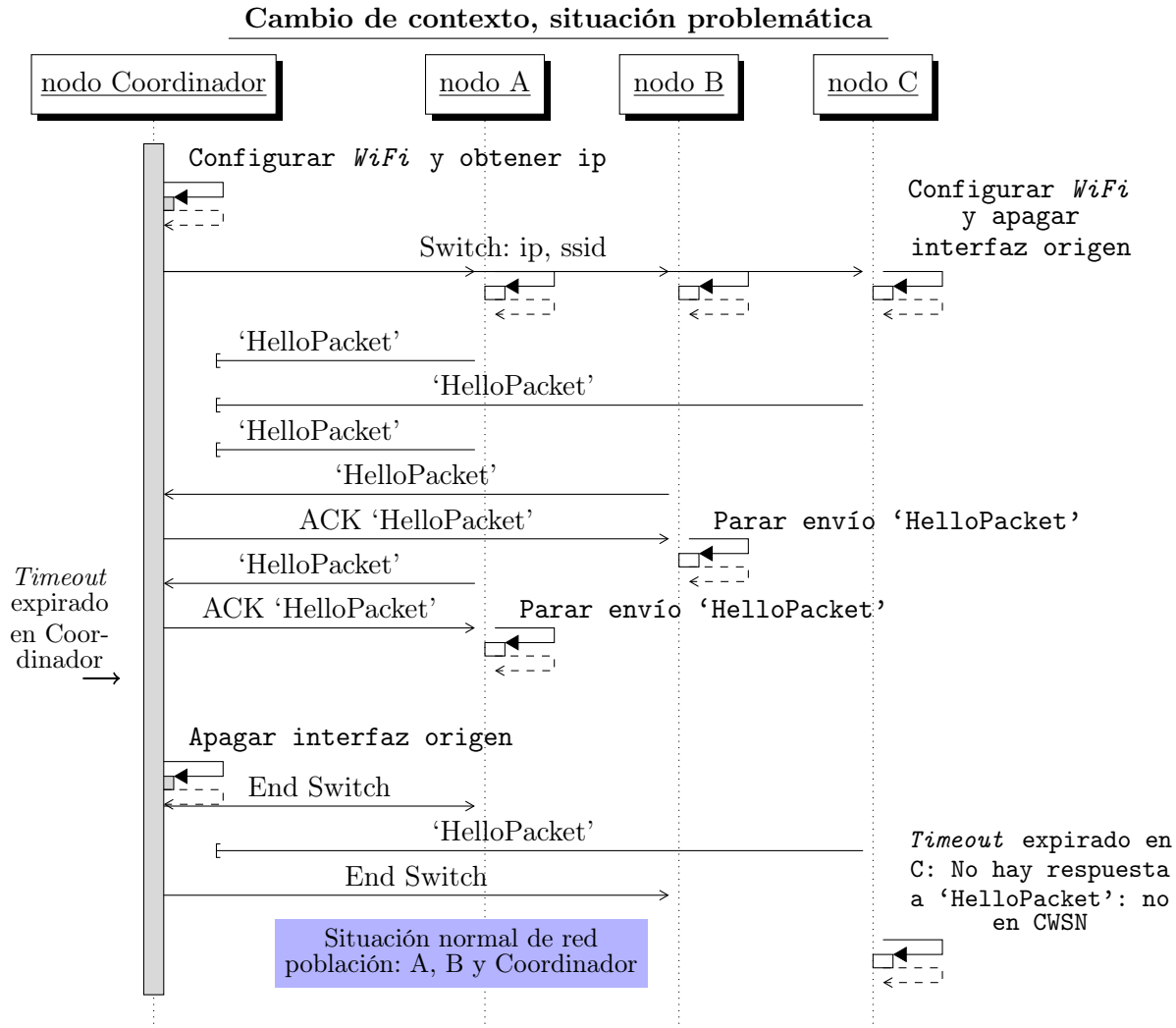

Figura 1.11: Ejemplo cambio de contexto hacia *WiFi* con pérdida de nodos

1.3.8.2. Cambio de contexto con interfaz destino Bluetooth

Código invulcrado

Service: coordinatorCognitiveTask(), mayTakeAdvantageOfBT(), contextChangeToBT(), SetupBTinterface (asyncTask), mHandlerBt, tellBTisUP(), scheduleEndSwitchBTRunnable (future) scheduleSetupBTRunnable (future) , sendEndSwitchMessage(), endContextualChange()

Cuando se produce un cambio de interfaz a *Bluetooth* el coordinador al igual que en el otro cambio de contexto obtiene una lista de quién está en la red y se marcan como inactivos. Comunicamos hacia las aplicaciones el nuevo estado de la red y procedemos a enviar un mensaje de señalización que marque a los demás nodos el evento de cambio

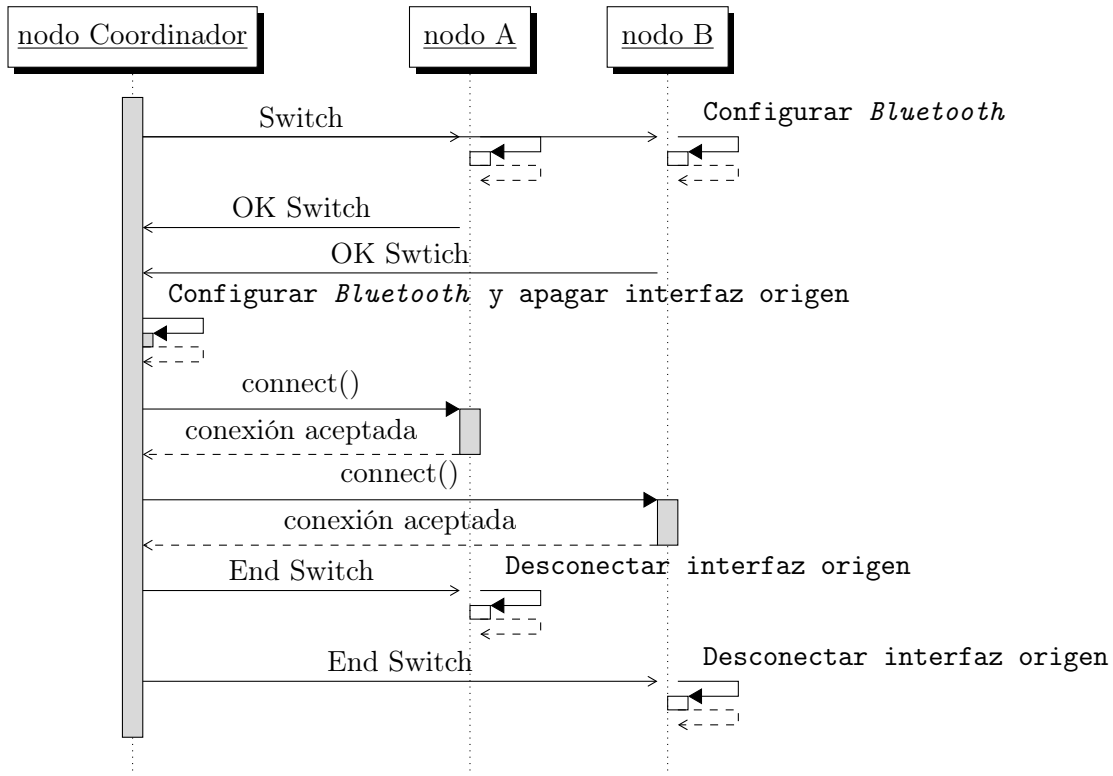

Figura 1.12: Ejemplo cambio de contexto hacia *Bluetooth*

de contexto hacia *Bluetooth*, este mensaje tiene la disposición mostrada en la tabla 1.38

Una vez enviado, programamos la tarea de configurar la interfaz de *Bluetooth* en el coordinador con un margen de tiempo suficiente para obtener las respuestas de los nodos normales, que al recibir la señalización configuran su interfaz *Bluetooth* e informan a sus aplicaciones del nuevo estado. Una vez levantada, comunican al coordinador través de la interfaz en curso (*Bluetooth* aún no es efectiva para envíar mensajes, ya que entre otras cosas no hemos establecido ninguna conexión con el coordinador) que la interfaz destino está lista para aceptar conexiones, para ello utilizamos el mensaje ya visto en la tabla 1.36, reutilizamos este mensaje de confirmación que no lleva ninguna información especial y que por el contexto le damos el significado de estar preparado (en la otra ocasión el contexto le daba el significado de confirmación de entrada en la red).

Este mensaje le sirve al coordinador para saber que el nodo ya está listo y cuando recibe todas las confirmaciones (o salta el timeout programado anteriormente) procede a configurar su interfaz de *Bluetooth* pasándole la lista de nodos que obtuvimos. La configuración de la interfaz incluye una ronda de conexiones a los nodos que formaban la red antes del cambio, que es pasada como parámetro. Una vez terminado ejecuta los mismos pasos descritos en la sección anterior para terminar el cambio de contexto. Salvo que en el mensaje de señalización de fin de cambio de contexto (ver tabla 1.37) la representación de cada nodo no incluye el campo “Dirección IP”.

Por su parte el nodo normal al configurar su interfaz *Bluetooth* y después de enviar

Estructura mensaje SWITCH hacia *Bluetooth*

Tipo Valor “SIGNALING” del enumerado “MessageType”.

Desde Una *cadena de caracteres* con la direción MAC de *Bluetooth* del nodo.

Mensaje de señalizacion Un contenedor que contiene al mensaje del tipo que se indicó en el campo “Tipo”:

- **Señal** Valor “SWITCH” del enumerado “SignalType”.
- **Mensaje de SWITCH** Un contenedor que contiene al mensaje del tipo que se indicó en el campo “Señal”:
  - **Interfaz destino** Valor “TO_BT” del enumerado “ToInterface”.

Tabla 1.38: Campos del mensaje SWITCH hacia *Bluetooth*

la confirmación al coordinador de que está listo, programa una tarea para salir de la situación de cambio de contexto por si no recibe ninguna conexión entrante o por si no recibe el mensaje de señalización de fin de cambio de contexto. Así se garantiza que ningún nodo se quede en *deadlock* y el flujo siempre se ejecuta de principio a fin.

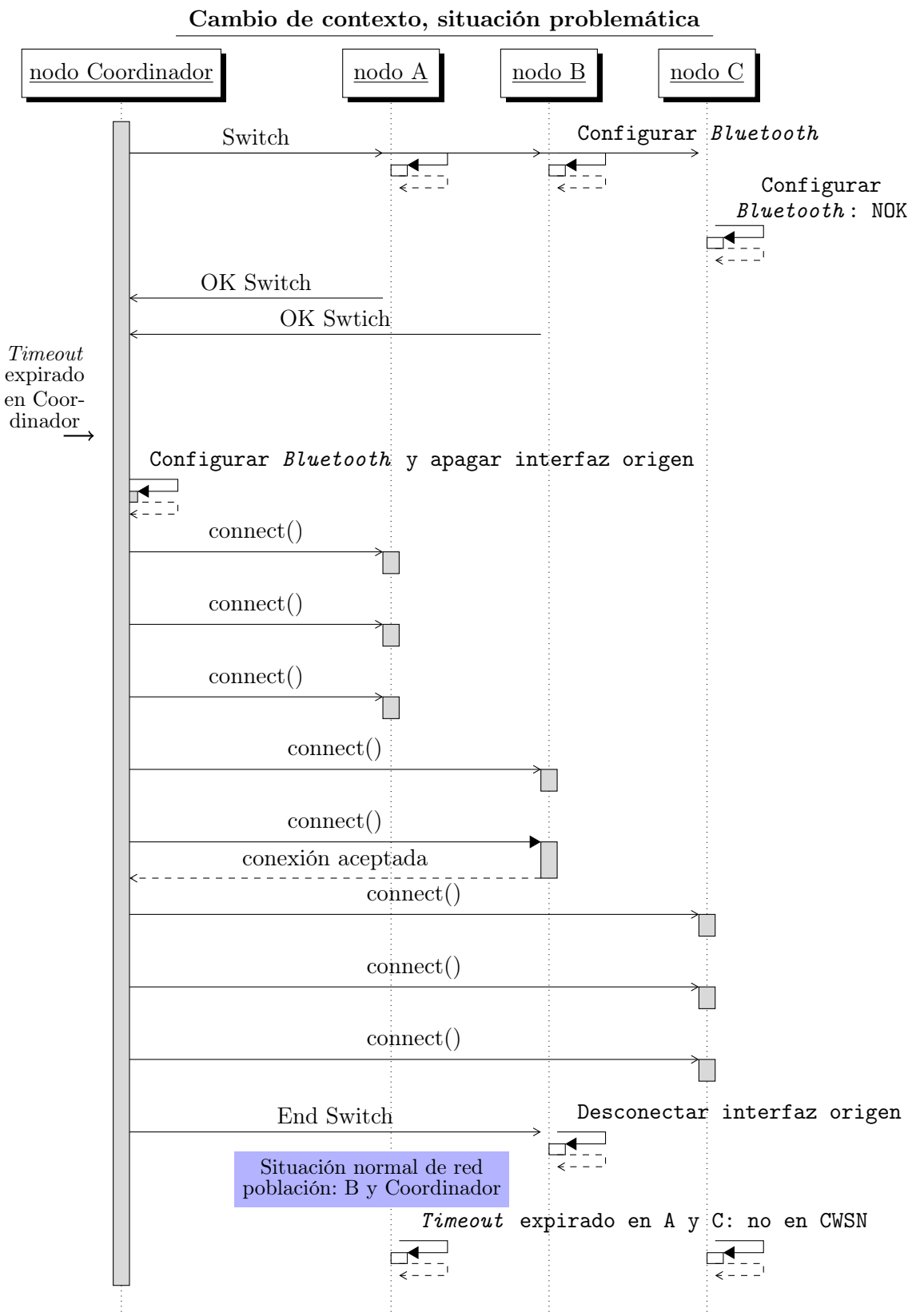

Figura 1.13: Ejemplo cambio de contexto hacia *Bluetooth* con pérdida de nodos

1.4. Aplicación utilitaria servicio cognitivo

Para hacer uso y probar las características cognitivas de nuestra arquitectura se ha diseñado una aplicación que se monta sobre el módulo o servicio cognitivo. La aplicación desarrollada además de cumplir la API en su totalidad (incluida la denominada parte oculta) permite acceder a los parámetros de la politica cognitiva y los timeouts utilizados en los procesos de red.

La aplicación simula el comportamiento de un sensor cuyo valor lo introduce el usuario y monitoriza el estado del servicio gracias a los mensajes de información que se envían desde el propio servicio o módulo cognitivo. Se estrutura en tres secciones: la principal que da cabida al objetivo de la aplicación y permite en caso de *Bluetooth* la conexión manual al nodo. En una segunda sección se muestra (si el usuario lo requiere desde la configuración) unas funciones de *test* entre las que podemos enumerar:

Petición, respuesta Permite dependiendo del tipo de nodo, guardar una respuesta en el caso de nodos normales o pedir esta información para los nodos coordinadores. Con esta función se habilita un mecanismo de petición de información muy útil para la arquitectura cognitiva para la compartición de datos de sensado.

Encolamiento de mensajes Debido a que la aplicación actualiza de forma instantánea la lista de nodos al recibir un evento de cambio de contexto, se inabilita la elaboración de mensajes de datos que entregar al servicio debido a que no es posible construir una lista de destinatarios válida puesto que el único valor seleccionable por el usuario es '0' cuyo significado es la ausencia de nodos. Al intentar entregar un mensaje de datos al servicio con este valor, la validación que existe en la aplicación impide finalmente su entrega.
Por este motivo no es posible generar un mensaje válido que se entrege al servicio en el momento que produzca su encolamiento, por lo que la aplicación permite gracias a un *shortcut (workaround)* el forzado para el encolamiento. Al pulsar el botón diseñado a tal efecto, se nos pregunta a través de una cadena de diálogos el mensaje a encolar y a que destinatarios debe dirigirse.

Vaciado de la cola de mensajes El vaciado de la cola de mensajes pendientes de envío se produce al efectuarse un cambio de conexto, una vez las funciones de red están nuevamente activas, si se quiere forzar el vaciado de esta cola por motivos de *test* se ha habilitado un botón junto al de la función anterior a tal efecto.

Envío periódico datos sensor Una característica muy presente en redes de sensores es la que los sensores suelen enviar datos de forma periódica. Simulamos esta característica en la aplicación tras selecionar el periodo, los datos enviados consiste en una *cadena de caracteres* más un número que incrementamos en cada envio.

Por último, en una tercera sección que alberga la configuración. En ella podemos elegir:

Ajustes del servicio Tipo y nombre del nodo, periodo tarea cognitiva y valores de la política que regula los cambios de contexto (ver 2.2.7.3).

Ajustes de la aplicación Código de aplicación, papel del nodo y si el texto que se usa como valor del sensor debe borrarse después de su envío al servicio.

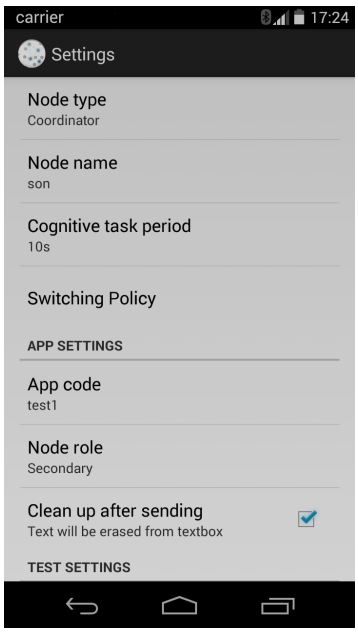

Figura 1.14: settings1

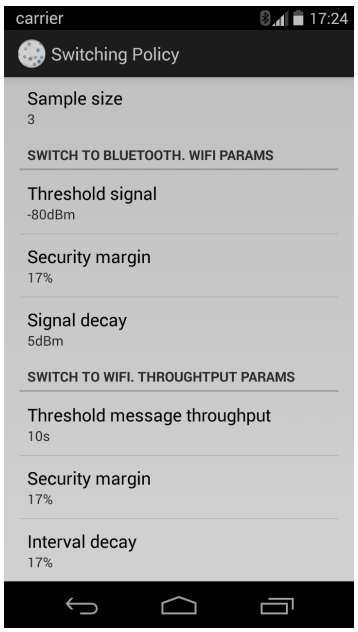

Figura 1.15: settings2

Ajustes de test Toda la configuración de la segunda sección como: si ha de visualizarse en la pantalla principal, con qué interfaz debe arrancar el servicio si la sección de *test* está activada y los *timeouts* utilizados para los *handovers*.

La aplicación proporciona validaciones adicionales para evitar el incumplimiento de la API o la interacción con el servicio si no es necesaria. No se permite:

- El envío de mensajes de datos si no está establecida una CWSN.
- El envío de mensajes no dirigidos a un nodo válido (tanto si es inactivo pues no se permite su selección al no estar presente entre los *items* que se presentan al usuario, como si el único valor que se presenta es la ausencia de nodos).
- La actualización vacía de la configuración. Es decir si no hay un cambio en la configuración no se traspasan las modificaciones necesarias al servicio para su actualización.

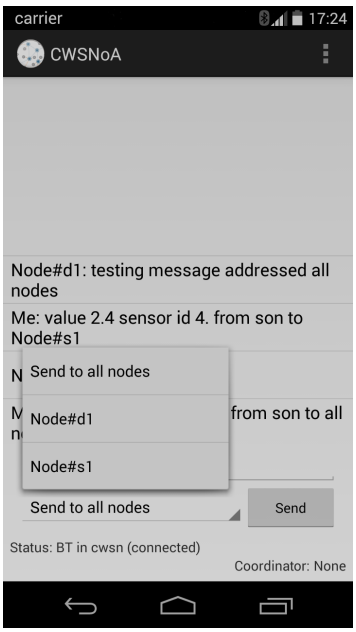

Figura 1.16: main ui

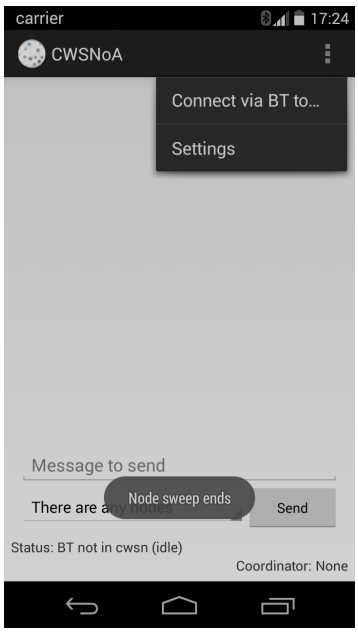

Figura 1.17: menu

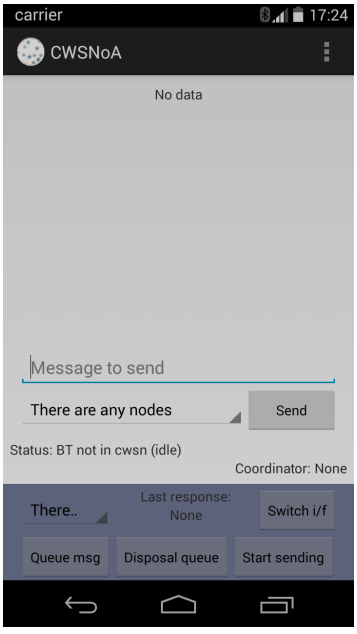

Figura 1.18: testing ui

Capítulo 2

Implementación

Cuando cortejas a una bella muchacha, una hora parece un segundo. Pero si te sientas sobre el carbón al rojo vivo, un segundo parecerá una hora. Eso es relatividad.

Albert Einstein

Este capítulo es complementario al de diseño, existe una delgada línea entre diseño e implementación que es muy fácil traspasar, quizás debido a que según cómo se mire la implementación puede interpretarse como el diseño de la siguiente capa a lo largo de éstas según bajamos hacia el *hardware*. En él se intenta explicar el cómo se ha hecho, es decir, qué componentes *software* se han utilizado para tal propósito.

2.1. Detalles de implementación

Para la implementación de los distintos bloques de la arquitectura se han elegido los mecanismos que mayor grado de independencia obtienen ayudando a estructurar los procesos en capas y tener puntos de intercomunicación entre éstos fuertmente marcados. La programación basada en objetos y el paso de mensajes entre distinas entidades ayudan a resolver las directrices de modularidad marcadas en diseño.

2.1.1. Interfaz aplicación servicio (cliente-servidor)

Como se vió en análisis **??**, de entre las formas de recrear la interfaz cliente-servidor necesaria para la confección de un *bound service*, escogemos aquella que se realiza a través de un *Messenger*. De esta forma la interfaz no declara métodos para su llamada desde el cliente, si no que la interacción con el servicio se basa en el intercambio de mensajes que son recogidos en su método *handleMessage()* declarado en su *Handler*. Por consiguiente tenemos una interfaz mucho más flexible y actualizable a cambio de estar abierta a errores (mensajes intercambiados sin fijarse a las reglas previamente declaradas), además de proveer de un entorno *thread-safe* al encolar las solicitudes de los diferentes clientes y ser entregadas al servicio en orden una vez que se haya atendido a la petición mediante la hebra que maneja el *Handler*. La interfaz o el objeto *IBinder* se construye gracias al método *getBinder()* del objeto *Messenger* creado gracias al *Handler* implementado en el servicio.

A su vez las aplicaciones deben declarar su propio *Handler* para poder recibir mensajes desde el servicio, para ello se deberá informar al servicio cúal es el objeto *Messenger* de la aplicación. Así habilitamos también en sentido contrario la comunicación habilitando al servicio que nos comunique datos en cualquier instante de tiempo, al igual estos envíos de información serán encolados por lo que la aplicación no debe preever mecanismos *multi-thread* para atender al servicio.

2.1.2. Módulo o Servicio cognitivo

El servicio se constituye en dos clases con una relación de heredamiento entre ambas. La clase padre `es.upm.die.lsi.pfc.CWSNoA.CognitiveLayer_common_Service` que intenta encapsular las partes que todo servicio cognitivo tendría que satisfacer:

- Métodos de inicialización de los componentes necesarios no específicos y su destrucción. (*onCreate(), onDestroy()*).
- Inicialización del *Handler* para la comunicación con las aplicaciones y manejo de mensajes de registro y no reconocidos.
- Obtención de la referencia al repositorio.
- Programación de la tarea periódica (optimizador) y la obtención de recursos para evitar que *Android* duerma al proceso en el trascurso desde la recepción de la alarma en un *BroadcastReceiver* hasta la llamada a *onStartCommand()* y la ejecución de ésta por completo.

Dejando las acciones específicas y propias de cada CWSN para la clase hija `es.upm.die.lsi.pfc.CWSNoA.CognitiveLayer_specific_Service`, entre las que destacamos:

- Implementación de los diferentes *Handlers* usados en la comunicación con las interfaces
- Implementación del resto de la arquitectura cognitiva: optimizador, ejecutor, políticas, acceso.
- Implementación de procesos de red: cambios de contexto, confeccionador e interpretador de mensajes de red, enrutado...

Realmente no hay una relación de *parent-son*, sino mas bien de *peer-to-peer* y las competencias tampoco están muy bien delimitadas, fallo del autor, pero si que es verdad que se consigue cierta encapsulación y nos ayuda a centrarnos exclusivamente en la programación del proceso cognitivo más que en los detalles necesarios para proveer del entorno necesario para llevar a cabo esa tarea.

Programación tarea Se basa en el sistema de alarmas de *Android* [4], este sistema es útil para realizar periodicamente operaciones sin tener en cuenta el ciclo de vida de la aplicación (en este caso el servicio cognitivo). Una alarma puede ser usada para lanzar *Services* en conjunción con *BroadcastReceivers*. Al dispararse la alarma gracias a la clase `android.app.AlarmManager` y efectuarse el *pendingIntent* descrito en ella, se ejecuta el método *onReceive()* del *BroadcastReceiver* (puesto en el *intent*) este método se ejecuta en la hebra principal y si dura demasiado tiempo *Android* matará el proceso por ello este método debe ser ligero y lanzar por ejemplo un *service*. Entre que se

termina de ejecutar el método y empieza a ejecutar el servicio (en este caso) ocurre un espacio de tiempo [5, p. 328-342] en que nuestro dispositivo puede dormirse. Por lo tanto debemos adquirir en el método *onReceive()* un *wakelock* que impida que el dispositivo pueda dormirse en este espacio de tiempo y liberarlo una vez se ha terminado de ejecutar las operaciones deseadas.
Para hacer este truco, llamamos a un método (que debe ser estático para poder llamarlo sin ser objeto) de nuestro servicio (alojado en *common*) que adquiere el cerrojo e inicia el servicio. Cuando un servicio es iniciado se crea el servicio si no estuviese ya iniciado o vinculado (*bound*) se ejecuta el método *onStartCommand()* una vez terminado no se produce a la destrucción hasta que se pare el servicio. Nuestro servicio permite ser vinculado para prestar conectividad en una CWSN y permite ser iniciado para realizar operaciones de forma periódica. La vinculación la realizan otras aplicaciones, sin embargo la iniciación la realiza el mismo servicio gracias a la sucesión de alarmas que se programa cuando se crea el servicio (método *onCreate()*), como consecuencia la extinción del servicio tiene que ser llevado a cabo por el mismo cuando detecte que no hay aplicaciones sobre él a las que dar soporte. El flujo de vida del servicio se ve alterado ya que al haber una iniciación, la falta de vinculación no detendrá al servicio, siendo preciso la llamada al método *stopSelf()* cuando sea oportuno (ver 2.2.4).

2.1.3. Repositiorio, capa de datos

Utilizamos la tecnología de base de datos de `Sqlite3` para implementar nuestro repositorio. La principal razón de esta elección es que se trata de una base de datos integrada en *Android* lista para usar que consume pocos recursos y se adapta bastante bien a casi todas las necesidades que pueda tener una aplicación móvil.

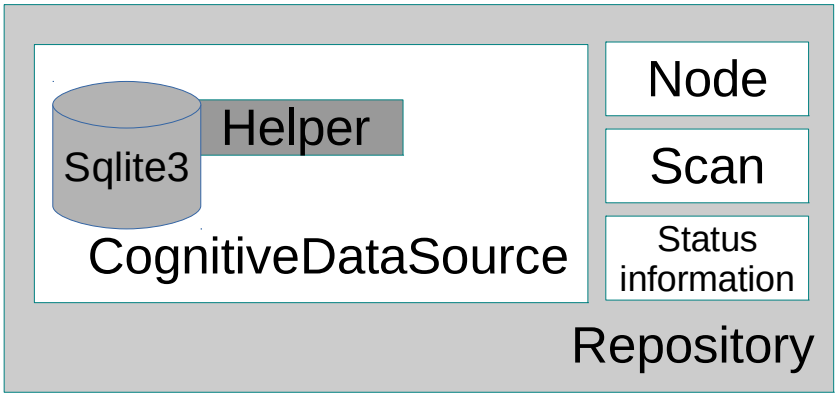

Figura 2.1: Repositorio, estructura interna

El modelo de capa de datos se basa en la propia base de datos `SQLiteDatabase` y un manejador para la creación, actualización, apertura y liberación de ésta: `MySQLiteHelper`, más una clase, que engloba a ambas y que abstrae aún más al servicio de la tecnología usada y de su tratamiento ya que ofrece la funcionalidad de introducir o hacer consultas sobre datos a través de métodos como si se tratase de un objeto más. Se trata de la clase `es.upm.die.lsi.pfc.CWSNoA.util.CognitiveDataSource`. *Android* nos provee de una API para la realización de *insert, update, remove* o *query* a través de métodos que

toman como parámetros los distintos datos de una sentencia o nos permite lanzar en crudo la sentencia de base de datos. Esto último tiene como problema que si la sintaxis cambia en el lenguaje SQL (*Structured Query Language*) nuestra sentencia puede verse afectada, sin embargo si se realiza de la primera forma somos inmunes a los cambios en el lenguaje debido a que la sentencia se construye en tiempo de ejecución en base a la tecnología que se posea en ese momento. La manera de proveer datos a estos métodos es a través de *arrays* de *cadenas de caracteres* en el caso de consultas o eliminaciones o utilizando el objeto `ContentValues` para nuevas inserciones o actualizaciones. El resultado lo obtenemos en un objeto `Cursor`, la clase abstrae al servicio al preparar los datos para elaborar la sentencia y entrega los datos obtenidos del cursor en forma de objetos primitivos básicos o más complejos englobandolos en objetos de clase como:

- `es.upm.die.lsi.pfc.CWSNoA.util.Node`
- `es.upm.die.lsi.pfc.CWSNoA.util.Scan`
- `es.upm.die.lsi.pfc.CWSNoA.util.StatusInformation`

2.1.3.1. Esquema base de datos

Vimos en 1.1.4 la información que necesitamos guardar, la estructuramos en tres tablas formado el siguiente esquema de la base de datos.

Tabla: Me		
PK	_id	Un entero auto incrementado
	currentInterface	Una *cadena de caracteres*
	state	Una *cadena de caracteres* no nula
	stopped	Un *boolean* no nulo
	periodTask	Un enterono nulo

*PK: Primary key, FK: Foreing key

Tabla 2.1: Estructura tabla Me

La tabla `Me` (tabla 2.1) contiene información acerca del estado del servicio, tiene un registro único que vamos actualizando conforme cambie la situación. El campo *currentInterface* contiene los valores 'bt' o 'wifi' y *state*: 'down', 'listen', 'connected'. El campo *stopped* nos indica que el servicio ha finalizado voluntariamente si tiene el valor '*true*' en el momento de inicializar el servicio. La precisión del campo *periodTask* es de milisegundos.

La tabla `Nodes` (tabla 2.2) alberga la representación de todos los nodos conocidos. Con el campo *active* discernimos si forman parte de la actual CWSN. Esta tabla nos permite crear el mapa de red y construir las diferentes rutas para el encaminamiento de paquetes gracias al campo *id_ coordinator*. Notar que el campo *macBT* se necesita que sea único para pero no se declara como clave primaria de la tabla puesto que es mejor tener un índice dedicado a ello (*id_ node*). Al crear esta tabla rellenamos el primer registro con los valores del propio nodo, así por diseño, el identificador '1' hace referencia siempre a sí mismo.

Tabla: Nodes		
PK	id_node	Un entero auto incrementado
	name	Una *cadena de caracteres* no nula
	role	Una *cadena de caracteres* no nula
	type	Una *cadena de caracteres* no nula, defecto: 'n'
	macBT	Una *cadena de caracteres* no nula, única
	ipAddress	Una *cadena de caracteres*
	id_coordinator	Un entero
	active	Un *boolean* no nulo, defecto: '0' se interpreta como '*false*'

*PK: Primary key, FK: Foreing key

Tabla 2.2: Estructura tabla Nodes

Tabla: Scans		
PK, FK(PK Nodes)	id_node	Un entero no nulo
PK	id_scan	Un entero no nulo
	rssi	Un entero
	throughput	Un *double*
	timestamp	Un *datetime*, defecto CURRENT_TIMESTAMP

*PK: Primary key, FK: Foreing key

Tabla 2.3: Estructura tabla Scans

La tabla Scans (tabla 2.3) hace referencia a los datos de sensado del entorno. La clave primaria está formada por el par (*id_node*, *id_scan*), de tal forma que no se permiten valores repetidos de este par. Vinculamos esta tabla con la tabla Nodes gracias a la clave foránea *id_node* que hace referencia al campo *id_node* de Nodes de tal forma que no se permite introducir datos del entorno de nodos que no existan y borrar nodos en la tabla Nodes si existen datos en la tabla Scans para no dejar huérfano el dato. La manera de introducir nuevos valores implica una consulta anterior para averiguar cual es el número de *scan* que corresponde al nodo al que queremos vincular el dato. Este proceso está encapsulado en el método que se publica en la capa de datos siendo transparente al servicio.

2.1.4. Mensajes enviados por la red

Estos mensajes son resultado en última instancia de InterfaceManager que es el punto de salida del módulo cognitivo hacia los 'controladores de interfaz' para su envío por el canal de datos o por el VCC. La implementación esta basada en *Protocol-Buffers* de Google [6]. Los mensajes son construidos utilizando IDL (*Interface Definition language*), un archivo donde definimos la estructura de estos mensajes. *Protocol-Buffers* utiliza un entero para cada campo para evitar codificar el nombre del campo y así serilizar estructuras de datos de manera más eficaz. La manera de codificar este entero varia por la misma razón: desde el 1-15 se codifica con un *byte*, del 16-2047

con dos y así en adelante. *Protocol-Buffers* define una entidad llamada '*Message*' que puede albergar campos de distintos tipos: int32, bool, bytes... tipos más complejos como enumerados u otros mensajes anidados.
Con esta IDL que definimos en la clase `es.upm.die.lsi.pfc.CWSNoA.util.Message` con extensión `.proto`, *Protocol-Buffers* nos genera (gracias al compilador `protoc`) un *parser* y todos los métodos necesarios para construir y serializar las estructuras de datos. Usamos la sentencia `option optimize_for = LITE_RUNTIME;` en nuestro archivo IDL para indicar que nuestra maquina es limitada y así usar los menos recursos posibles a cambio de perder otras características como la rapidez.

Para facilitar el *parsing* de los diferentes mensajes, se ha definido dos grandes mensajes para evitar anidar mensajes que no comparten un significado, lo que haría poco claro el modelo. El primer mensaje engloba todos los mensajes intercambiados en la CWSN, [`NodeMessage`] y el otro engloba a los paquetes que se envían sobre *WiFi* para el registro y desregistro en la red, [`WifiPacket`]. Empezamos a interpretar un mensaje asumiendo (por probabilidad) que es del tipo `NodeMessage` si obtenemos un error del tipo `InvalidProtocolBufferException` significa que el mensaje recibido es del otro tipo. Para saber qué tipo de mensaje está contenido evaluamos `type` y `packetType` respectivamente, estos campos no deben ser codificados con el mismo número, de ser así el proceso de *parser* puede confudir ambos tipos de mensaje.

En la figura 2.2 se plasma un mapa completo de la estructura de anidación de los diferentes mensajes que se han visto en los procesos de red (sección 1.3 del capítulo de diseño). En color **turquesa** se pintan el nombre de los campos que son opcionales y en **negro** aquellos que son obligatorios, los denominados con el prefijo *list* en el tipo, aceptan más de un valor convirtiéndose en una lista del tipo denominado. Aunque se nos permite anidar más de un tipo de mensaje, la manera de interpretarlo está guidada por los campos donde se especifica qué tipo de mensaje contenemos, por ello cada mensaje anidado está marcado como opcional pero siempre debemos asegurar que está presente el mismo tipo de mensaje que el especificado en el campo que usamos como guía.

2.1.5. Interfaz servicio controladores

Esta comunicación está fuertemente marcada y se basa también en el paso de mensajes. Esto nos ayuda a colocar la funcionalidad en cada parcela e intercambiar exclusivamente y de forma estandarizada la información necesaria para llevar a cabo un proceso.

En el servicio implementamos tantos *Handlers* como controladores de interfaz tengamos, que pasamos como referencia a éstos. El controlador por tanto tiene una forma de remitir información de manera estandarizada sin conocer más datos del servicio, el cúal si que puede lanzar los procesos (llamar a métodos del controlador) al estar en un nivel jerárquico superior. El servicio recoge los diferentes mensajes enviados por el controlador que son entregados en orden y cuando se haya atendido a la petición anterior al basarse en la misma implementación que la interfaz aplicación servicio. Estos mensajes aglutinan tanto el evento que se produzca en la red como el estado de la interfaz.

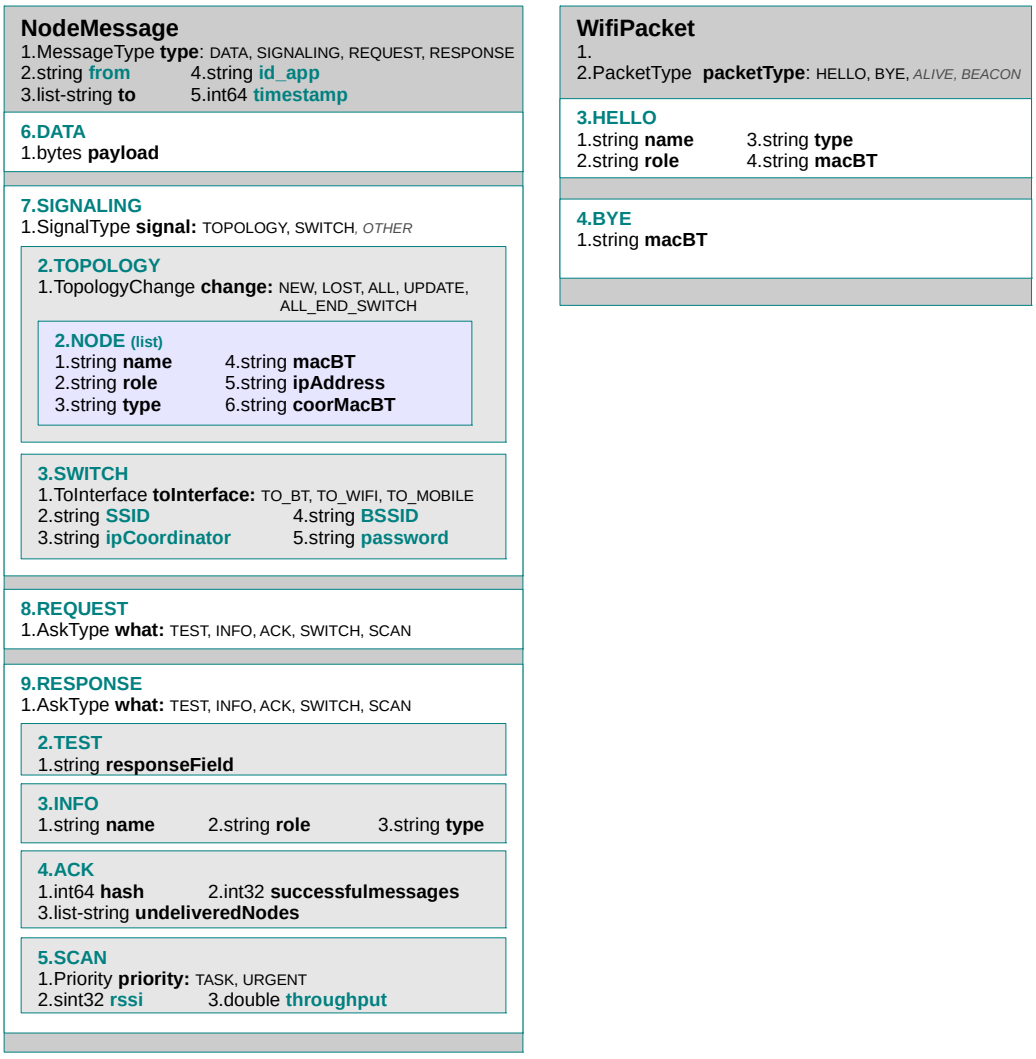

Figura 2.2: Estructura anidada de mensajes de red sobre *Protocol-Buffers*

2.1.6. Controlador interfaz WiFi

Está formado por:

- La clase `es.upm.die.lsi.pfc.CWSNoA.WifiController` que maneja las conexiones entrantes sobre IP y centraliza todas las funciones de la interfaz para enviar y recibir datos, así como el descubrimiento y salida de la red.
  Contiene dos hebras encargadas de alojar *sockets* utilizados en la comunicación y un gestor de estados (*setState()*) que va recabando y enviando la información al servicio sobre el estado de la interfaz. Necesitamos alojar esto en una hebra pues son llamadas bloqueantes, no acotadas en el tiempo y el tiempo de ejecución no está delimitado, es decir, una vez salgamos del bloqueo vamos a querer bloquearnos otra vez para no perder ninguna conexión.

- El *ServiceIntent* `es.upm.die.lsi.pfc.CWSNoA.util.SendoIP` que maneja las conexiones salientes sobre IP. En este caso la implementación es sobre un servicio en vez de una hebra propiamente dicha. Esto es así porque los servicios están pensados para realizar una tarea sin que el componente que lo inicializó influya en su ciclo

de vida y al revés. El envío de datos por la red es un ejemplo de esto y al ser una actividad acotada en el tiempo (no sabemos cuanto va a tardar, pero tiene delimitado un final) podemos encomendar esta tarea a un *serviceIntent*, un servicio que se ejecuta en su propia hebra y que se autodestruye cuando ha terminado de ejecutar la tarea. Además las peticiones simultáneas a este tipo de servicios son encoladas y entregadas una vez ha terminado la anterior, asegurándonos que sólo tenemos un *socket* abierto para el envío de datos y que no hay condiciones de carrera en el envío de distintos datos a distintos participantes.

- La *AsyncTask* encargada de inicializar el `WifiController` y del manejo de la API *WiFi* de *Android*. La tarea: `es.upm.die.lsi.pfc.CWSNoA.CognitiveLayer_specific_Service.SetupWifiInterface` está ubicada dentro del servicio aunque no se aprovecha del *scope* en el que está, por lo que puede declararse independientemente en su archivo y funcionar de la misma manera.
- Un *broadcastReceiver* `es.upm.die.lsi.pfc.CWSNoA.StatusReceiver_wifi` encargado de recoger los eventos de conexión o cambios para sincronizar los procesos en el controlador. En la configuración de la interfaz como proceso de red explicaremos los motivos del porqué de una *AsyncTask*.

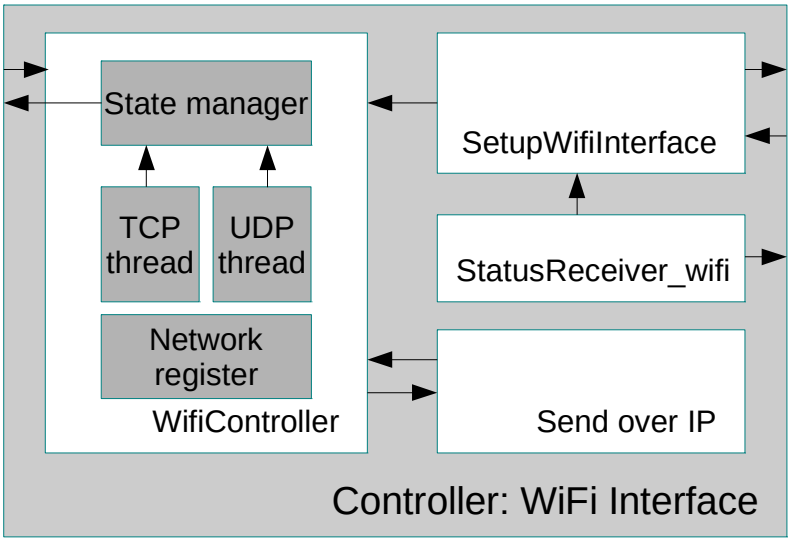

Figura 2.3: Controlador WiFi, estructura interna

El servicio inicializa el controlador de la siguiente manera:

1. Inicializar el *BroadcastReceiver* y añadir los filtros pertinentes para registrarse a los eventos deseados. Lo realizamos en el método *onCreate()* y limpiamos este registro en el *onDestroy()* para evitar *memory-leaks*.
2. Crear el objeto `WifiContoller` pasándole la referencia del *Handler* para poder enviarle mensajes. Este objeto necesita ser configurado con la llamada al método *start()*
3. Lanzar la *AsyncTask* asociada a la configuración de la interfaz, como veremos este proceso es costoso por lo que se tiene que ejecutar fuera de la hebra principal

(la llamada *UI-Thread*, encargada de recoger la interacción del usuario) evitando la pérdida de rendimiento y errores tipo ANR (*Android Not Responding*). A su vez esta tarea termina de configurar el `WifiController` en la modalidad deseada.

2.1.7. Controlador interfaz Bluetooth

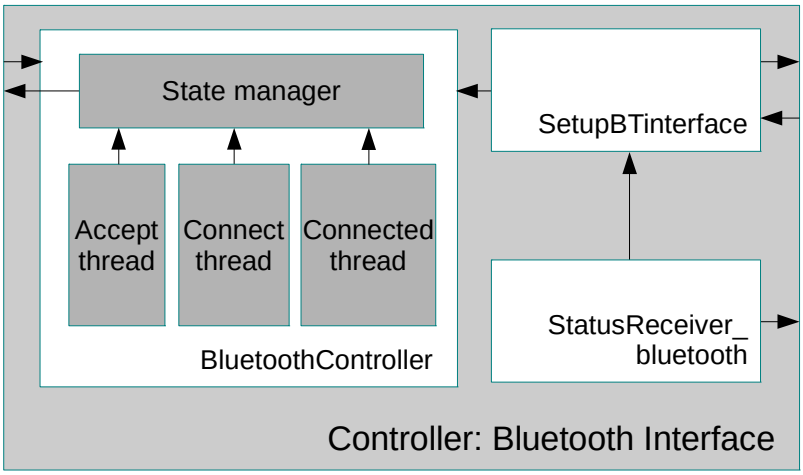

Figura 2.4: Controlador Bluetooth, estructura interna

Como se ve ambos controladores tienen una arquitectura similar. El de *Bluetooth* está formado por:

- La clase `es.upm.die.lsi.pfc.CWSNoA.BluetoothController` manejadora del establecimiento y mantenimiento de las conexiones sobre un canal RFCOMM. Al igual que su homóloga implementa por la misma razón hebras que gobiernan los detalles de la comunicación. En concreto se usan tres tipos de hebras como veremos en la configuración de la interfaz, dos de ellas para conseguir el *socket* sobre el que enviar y recibir datos y otra para alojar y controlar a éste.

- La *AsyncTask* para la configuración de la interfaz y del objeto `BluetoothController`: `es.upm.die.lsi.pfc.CWSNoA.CognitiveLayer_specific_Service.SetupBTinterface`, imitando las mismas líneas marcadas en el controlador de *WiFi*.

- Un *broadcastReceiver* `es.upm.die.lsi.pfc.CWSNoA.StatusReceiver_bluetooth` encargado de recoger los eventos de encendido o apagado, para sincronizar procesos.

De igual similitud el servicio debe seguir los mismos pasos para configurar este controlador. Los *BroadcastReceiver* los tiene que inicializar el servicio para tener comunicación directa con éste, ya que necesita saber de los eventos ocurridos para distintos propósitos. Así evitar meter una capa intermedia al referenciarlos directamente desde el servicio, lo que nos permite su observación a lo largo de toda la vida de éste. Si ligasemos su vida a la del controlador sólo serían observables cuando la interfaz en uso sea la de la interfaz en cuestión, ya que por motivos de eficiencia y minimización de recursos el resto del controlador se libera al no ser utilizado.

2.2. Implementación de procesos de red

En esta sección relataremos los detalles de implementación de ciertas partes de la arquitectura cuya explicación viene más a colación de la mano del proceso que en la su bloque.

2.2.1. Registro de una aplicación en el servicio cognitivo

Código invulcrado

Service: m2service (manejador de mensajes entre servicio y aplicaciones)

En el registro de la aplicación en el servicio, guardamos el *Messenger* de la aplicación (que viene informado en el campo *replyTo* del primer mensaje de *handshaking*) en un *HasMap* llamado `mClients` cuya clave es la representación en forma de *cadena de caracteres* del objeto *Messenger* obtenida a a través del método *toString()*, también registramos su código de aplicación en otros dos *HashMap*:

- `mOut_app`: nos relaciona la clave de `mClients` con el código de aplicación. La aplicación al enviarnos la petición (en la que va incluida su *Messenger*) recuperamos su código de aplicación (gracias al *Messenger*) para poder marcar el mensaje de datos saliente. Sin necesidad de que la aplicación nos informe en cada petición de su código de aplicación.
- `mIn_app`: relaciona el código de aplicación con la clave `mClients`. Al recibir un mensaje leemos el código de aplicación que viene marcado, gracias a este *HashMap* obtenemos la referencia a `mClients` donde obtenemos el *Messenger* para poder entregar el mensaje de datos entrante a la aplicación correspondiente.

2.2.2. Configuración de la interfaz de comunicación

La configuración de cada interfaz de comunicación se realiza en una *AsyncTask*: clase diseñada por el sistema *Android* [7]. La peculiaridad de este tipo de tarea es que el núcleo central de ésta se ejecuta en una hebra independiente que es arrancada, manejada y destruida por el propio sistema *Android*, liberando al programador de aplicaciones de esta tarea. Además provee de mecanismos de compartición de información, tanto al principio, como en plena ejecución y también tras la finalización de la tarea entre la *UI-Thread*, desde donde arrancamos la tarea y la hebra donde se ejecuta propiamente ésta, haciendo una transición más fluida. Por esto último ha sido elegida en favor de un *ServiceIntent*.

2.2.2.1. WiFi: servicio y controlador

Código invulcrado

Service: SetupWifiInterface (AsyncTask)
WifiController: stop(), start() / StatusReceiver_wifi (BroadcastReceiver)

La esperas señaladas en 1.3.2 consistentes en levantar la interfaz *WiFi* o conectarse a una red *WiFi* se realizan mediante esperas de consulta activa o *polling* cuya variable de control es manejada en el *BroadcastReceiver* de esta interfaz: `es.upm.die.lsi.pfc`

`.CWSNoA.StatusReceiver_wifi`, gracias a una referencia de *callback* que nos permite acceder a las variables deseadas. En el primer caso el anuncio que esperamos tiene la *action* `WIFI_STATE_CHANGED_ACTION` cuyo valor que esperamos es `WIFI_STATE_ENABLED`, la variable de control usada es un booleano llamado `wifiON`. Para la segunda espera esperamos recibir un anuncio del tipo `NETWORK_STATE_CHANGED_ACTION` con el valor `CONNECTED`, la variable usada `waitUntilWifiGetConnected` accedida por métodos *getters* y *setters*.

Para encender la interfaz *WiFi* procedemos a invocar el método *setWifiEnabled(true)* de la clase `android.net.wifi.WifiManager`, es una llamada asíncrona que desencadena los procesos para encender el *driver* de *WiFi* y su anuncio mediante mensajes de *broadcast* del sistema.

Normal Necesita configurar un *ServerSocket* para permitir conexiones entrantes para recibir tráfico del coordinador. Para ello creamos el *socket* de la clase `java.net .ServerSocket` e intentamos vincularlo a un puerto con *bind()* pasandole como parámetro el puerto donde escuchar. Si este puerto está siendo utilizado para otros propósitos nos devuelve un error del tipo `IOException`. El valor de este puerto es una constante en el código (*Hard-coding*) cuyo valor es '7617' al ser un conocmiento compartido a priori, evitamos tener que transmitir este valor por la CWSN.

Coordinador También necesitamos escuchar tráfico proveniente de los nodos normales, por lo que desplegamos un *socket* que escuche en TCP al igual que el nodo normal. Además para poder registrar nodos en la red, dónde éstos no conocen a priori la configuración ni su esquema, necesitamos escuchar en direcciones tipo: *multicast* o *broadcast*.

Para ello necesitamos de un *MulticastSocket* para ampliar el rango de escucha más allá de nuestra propia dirección IP en la red. Registramos el *socket* en un grupo *multicast* a través de la llamada al método *joinGroup()* de la clase `java.net.MulticastSocket` (clase heredada de `DatagramSocket`) pasando como parámetro la dirección *multicast* deseada y adquirir un cerrojo para que este tipo de mensajes en la cola de *WiFi* no sean descartados y sean entregados en el *socket*, para ello llamamos a *createMulticastLock()* de la clase `android.net.wifi.WifiManager`. Para escuchar en la dirección de *broadcast*, llamamos al método *setBroadcast()* de la misma clase, pasando como parámetro '*true*'. El puerto de escucha de este *socket* UDP es el número siguiente a dónde escuchamos en TCP.

También necesitamos adquirir un cerrojo para prevenir al *driver* de *WiFi* que deje de escuchar, el cerrojo encargado es el *createWifiLock()* que le pasamos como parámetro la constante `WifiManager.WIFI_MODE_FULL_HIGH_PERF` para indicar que se debe mantener la interfaz despierta, es decir, que se sigan capturando paquetes y se mantenga una latencia baja en la respuesta cuando la pantalla esté apagada.

Una vez terminada la configuración de la interfaz, ejecutamos en el método *onPostExecute()* de la *AsyncTask* ejecutado sobre la *UI-Thread*, en vez de en segundo plano, el envío del mensaje de finalización del *handshaking*.

2.2.2.2. Bluetooth: servicio y controlador

Código invulcrado

Service: SetupBTinterface (AsyncTask)

BluetoothController: stop(), start(), connect(), setState() / StatusReceiver_bt (BroadcastReceiver)

El controlador maneja tres tipos de hebras: la que acepta conexiones (*AcceptThread*), la que lanza la conexión (*ConnectThread*) y la que mantiene la conexión entre el esclavo y el maestro (*ConnectedThread*), de este tipo podemos tener más de una.

Un nodo normal necesita mantener una hebra del tipo *AcceptThread* en todo momento que no esté conectado a un nodo coordinador y que la interfaz en curso sea *Bluetooth*. Cuando está conectado al coordinador, mantiene una instancia de *ConnectedThread*.

El nodo coordinador necesita mantener tantas instancias de la hebra *ConnectedThread* como esclavos tenga conectados, estás hebras se guardan en un *HashMap* cuya clave es la dirección MAC de *Bluetooth* del dispositivo con el cual se establece la comunicación. Estas hebras se crean cuando, la hebra *ConnectThread* consigue establecer la conexión. La clave nos permite acceder al método *write()* de la hebra correspondiente cuando el serivio nos envía datos y la lista de direcciones MAC para volcar el mensaje a través de la red y llegue al destino oportuno.

La espera para levantar la interfaz *Bluetooth* también se realiza mediante *polling*, cuya variable de control que permite salir de la espera activa se maneja en su *BroadcastReceiver* (`es.upm.die.lsi.pfc.CWSNoA.StatusReceiver_bluetooth`) cuando llegan anuncios del tipo `ACTION_STATE_CHANGED` con el parámetro `STATE_ON`.

Para levantar la interfaz *Bluetooth* obviamos la recomendación de *Android* que incita a preguntar al usuario a través de un diálogo para la activación del *Bluetooth*. Como un requisito de la CR (*Cognitive Radio*) es ser transparente al usuario, utilizamos la llamada asíncrona del método *enable()* de la clase `android.bluetooth.BluetoothAdapter`. Esto desencadena su encendido y anuncios posteriores del sistema informando que ya está levantada que capturamos gracias al *BroadcastReceiver* de *Bluetooth*.

El método *connect()* arranca una nueva hebra tras invocarse para asegurarse de que su procesamiento no bloquee al resto del programa. La razón de asegurar una ejecución en segundo plano, reside en que este método es llamado en dos puntos de la arquitectura: al configurar la interfaz y al recibir un mensaje de *workaround* para conectarse de manera manual a un nodo (ver sección:1.2.10), en el primer caso este poceso ya está en segundo plano, pero en el segundo, si no se ejecutase en su propia hebra, se retrasaría el procesamiento de los mensajes encolados en el *Handler* del servicio con las aplicaciones sin necesidad ya que no hay que devolver ningún resultado.

La llamada a este método comienza con la interrupción de la hebra tipo *ConnectThread* (la que lanza la conexión) si estuviese ejecutándose por una llamada previa inacabada. Tras esto lanzamos una nueva instancia de la hebra y esperamos a que termine con la instrucción de Java *join()*, por el mecanismo interno de estados del controlador si la conexión se ha producido con éxito se habrá modificado un *flag* que nos permite terminar la ejecución del método, si no, reintentamos hasta que alcancemos el máximo de intentos (pasado como parámetro).

Mientras tanto la *AsyncTask* donde está alojado el proceso nos habrá esperado mediante espera activa, cuya salida está garantizada por los *flags*: `flagNewDevice` y `numFailConnection`, accesibles a través de los métodos *getFlagNewDevice()* y *getNumFailedConnection()* del servicio.

Una vez hemos terminado de lanzar las conexiones a los nodos indicados finalizamos la tarea en segundo plano, dejando la ejecución del envío del mensaje de finalización del *handshaking* en la *UI-Thread*.

2.2.3. Registro de un nodo en la red

Como vimos en diseño, la interfaz condiciona a quién tiene que dar el primer paso, esto se debe a restricciones de implementación.

2.2.3.1. WiFi

Código invulcrado

WifiController: SendHelloPacket (AsyncTask), UDPlistener (Thread), stopSendingHelloPacket(), setState()
Service: mHandlerWf, sendInfoNewNodeOnNetwork()
Database: eventNewDeviceIPCoordinator(), newDeviceEventNormal(), modifyNode()

Una red *WiFi* tiene más flexibilidad para crear la tipología de red que se requiera, podemos tener desde conexiones (condicionado al tipo de éstas) PPP (*Point-to-point*) hasta *multicast* o *broadcast* aunque por debajo tengamos un AP que enmascare la tipología utilizada. El tipo de conexiones también puede ser elegido mediante la capa de transporte, para el registro de un nodo normal en la red utilizamos un *DatagramSocket* para enviar paquetes UDP. La dirección a la que van dirigidos estos paquetes se alterna entre la dirección *broadcast* de la red a la que estemos conectados y una dirección de *multicast*: '224.2.76.24' para por motivos de diseño intentar incrementar el éxito de llegada e interpretación de estos paquetes, debido a fallos debido a la corrupción en los datos del paquete cuya notificación no llega al *socket* o debido a la menospreciación por parte del AP o al propio sistema *Android* que no permita al *driver* de *WiFi* estar siempre escuchando en la dirección de *broadcast*, para ello, enviamos una ráfaga de 20 paquetes espaciados 800 milisegundos, estos valores garantizan una buena acogida del nodo en la nueva red sin extender la latencia del proceso demasiado en caso de fallo. La alternancia de estos paquetes es 1 de cada 5 van dirigidos a la dirección de *multicast*, el resto a la de *broadcast*. El envío de estos paquetes se realiza, al ser un proceso largo, en segundo plano implementado como una *AsyncTask*.

2.2.3.2. Bluetooth

Código invulcrado

BluetoothController: connected(), setState()
Service: mHandlerBt, sendInfoNewNodeOnNetwork()
Database: newDeviceEventNormal(), modifyNode(), updateInfoDeviceCoordinator()

La única tipología de red permitida en una red *Bluetooth* como vimos en **??**, es la de estrella, formada por un maestro y varios esclavos (las puntas de la estrella), siguiendo este modelo podemos crear *piconets* [8] identificadas por un UUID (*Universally unique identifier*) necesario para entre otras cosas registrar los *sockets* que van montados sobre un canal *RFCOMM* [9]. Por ello es el nodo coordinador (el maestro) quién lanza las conexiones.

2.2.4. Salida de un nodo de la red

Tras recibir el mensaje de desregistro (ver tabla 1.17), procedemos a borrar la aplicación del mapa `mClients` y con él también las referencias cruzadas en los mapas `mOut_app` y `mIn_app`. Estos procesos se encapsulan en el método *removeClientknown-MessengerReply()* que necesita de la representación del objeto *Messenger* en forma de *cadena de caracteres* y *removeClientKnownAppCode()* que requiere del código de aplicación para efectuar el borrado.

Una vez no haya aplicaciones sobre el servicio procedemos a la destrucción del mismo como vimos en diseño en la sección 1.3.4, las acciones (tanto aviso como restauración de la situación inicial) a realizar las incluimos en el método *onDestroy()*, un método que llama *Android* automáticamente cuando no hay referencias al servicio, es decir todas las peticiones de *bind()* han sido correspondidas con sus *unbind()* y si ha habido una inicialización del servicio con *start()*, se ha llamado también al método *stop()*, en este caso este método lo llamamos desde el mismo servicio a través de *stopSelf()* cuando `mClients` pasa a no contener ninguna aplicación.

El envío, en el caso de *WiFi* del paquete *"ByePacket"* se realiza bajo una *AsyncTask* finalizando esta con la destrucción del controlador. El envío se realiza mediante un bucle de conexiones TCP a todos los nodos normales que haya en la red en caso de que el coordinador abandone la red. O sobre el envío bajo UDP (aunque están implementadas ambas formas) al coordinador si es un nodo normal quien abandona la red.

2.2.5. Actualización de parámetros en la red

Código invulcrado

Service: m2service, spreadNodeChangeOnNetwork(), updateInformationToCoordinator()

Como vimos en diseño tenemos un conflicto en la separación de capas, existe la necesidad de poder modificar todos los parámetros del nodo cognitivo para su adecuación y colaboración en cualquier CWSN. Tenemos dos opciones:

- No se permite actualizar estos parámetros si no es con la aplicación desarrollada en el mismo paquete que el servicio cognitivo, con lo cual el servicio cogería estos valores del archivo de preferencias (usando el objeto *Preference* [10]) ubicado en el mismo paquete que la aplicación y cada vez que se requiera un cambio en el nombre, periodo de la tarea cognitiva o tipo de nodo, tener que abrir la aplicación para cambiar estas.
- La otra opción pasa por permitir que cualquier aplicación pueda cambiar estos valores, garantizando que estas actualizaciones no penalicen a otras aplicaciones que estén en ese momento en ejecución.

Nos hemos decantado por la segunda opción, por ello en diseño vimos que hay una manera especial de actualizar estos parámetros y un mecanismo para garantizar que no haya conflictos. Ello exige comunicar todos los parámetros del servicio a las aplicaciones cuando estas se registran y el almacenaje de estos parámetros en base de

datos para independizar el archivo de preferencias que compete exclusivamente a la aplicación desarollada a la vez que el servicio pero que este no debería tener acceso a él.

2.2.6. Intercambio de mensajes

La transmisión y recepción de datos se realizan a través del intercambio de mensajes definidos en la API. Podemos destacar tres acciones: el envío, la recepción y una mezcla de ambos reservada sólo al coordinador que es el re-envío o *forward* de mensajes.

2.2.6.1. Recepción de mensajes

Código invulcrado

Service: m2service, mHandler<controlador>, incomingContentNormal(), incomingDataMessageToApp()

Este proceso empieza al salir del bloqueo de la hebra en el controlador de la interfaz de comunicación en curso:

WiFi En este caso salimos del bloqueo que nos produce la instrucción *accept()* al aceptar una conexión del cliente para que nos envíe datos. En este caso empezamos a leer datos con hasta que no queden datos disponibles (con la sentencia '*available()* == 0' del *stream* de entrada obtenido del *socket* con la instrucción *getInputStream()*) o el *byte* leído sea '-1'.
En ese momento cerramos la conexión y quedamos bloqueados a la espera de una nueva.

Bluetooth Como la conexión está establecida, la instrucción que nos bloquea es precisamente la de leer. Al igual cuando terminamos de leer (no hay mas datos disponibles o hemos leido '-1') nos bloqueamos de nuevo en la instrucción de leer, no podemos cerrar la conexión pues es necesaria para intercambiar la información de salto de frecuencia que utiliza *Bluetooth* y que para el controlador (y el serivicio por ende) es totalmente transparente.

Entregamos los datos leídos al servicio que interpretará el tipo de datos que se trata.

2.2.6.2. Envío de mensajes

Código invulcrado

Service: m2service, builtDataMessage(), getNextNodes(), send(), mHandler<controlador>, outgoingContentProgress()
DataMessageQueue: add(), hashMessage(), changeStatus(), remove()

El envío comienza con el encolamiento del mensaje como vimos en diseño, la manera de encolar éstos reside en clase `es.upm.die.lsi.pfc.CWSNoA.util.DataMessageQueue`, se trata de una cola FIFO implementada sobre un *LinkedHashMap*. La razón de no usar un objeto *queue* puro es que necesitamos acceder a cualquier elemento en todo momento debido a la heterogeneidad de estados en que pueden encontrarse los elementos de ésta.

El *LinkedHashMap* tiene como clave un tipo *long* que es el *hash* del mensaje. Para calcular el *hash* no podemos usar la función *hashcode()* de un *byte array* puesto que obtenemos valores distintos en cada invocación [11], por ello necesitamos realizar una función personalizada sencilla pues no preveemos grandes volumenes en esta cola por lo que la colsión es un riesgo bajo. Este *hash* lo calculamos de esta forma[1]:

```
1          long hash = 17;
2          hash = hash * timestamp;
3          hash = hash + 3 * from.hashCode() >> 1;
4          long tmp =  hash & from.hashCode();
5          hash = (hash ^ tmp);
6          hash = hash + 24 * payload.hashCode();
```

El objeto que guarda la clave en el *HashMap* es de la clase `MessageInQueue` (clase anidada de `DataMessageQueue`), cuyas varaibles de clase son:

1. Un enumerado del tipo `MessageState` cuyos valores pueden ser:
   - `PENDING_SEND`, se asigna este estado a los mensajes encolados que no pueden ser envíados en el momento de su recepción.
   - `SENDING`, estado asignado a los mensajes que permanecen en la cola mientras se está procediento a su envío (salida del nodo).
   - `PENDING_ACK`, estado que indica que el mensaje ha sido entregado en el siguente salto en la red pero falta confirmación de entrega en destino o destinos finales.
   - `OK` | Actualmente los mensajes que se han entregado correctamente se borran de la cola por lo que no se usa.
   - `NOK` | no usado. Útil por si queremos retransmitir el mensaje que falló en su entrega.
2. Un *byte array* que contiene el mensaje ya codificado listo para su envío.

Utilizando el *hash* del mensaje podemos recuperarlo para cambiar el estado de éste u obtenerlo para su envió a la aplicación tras producirse la confirmación de envío (ver tabla 1.9).

El envío de datos propiamente dicho se realiza en los controladores a través del método *write()*, estos métodos recogen, un *byte array* con el mensaje a enviar ya codificado , una lista de *cadena de caracteres* con las direcciones adecuadas a su interfaz y una *cadena de caracteres* en representación del origen del mensaje. El adaptamiento de los argumentos se realiza con la ayuda de dos métodos:

- *getNextNodes()*: este método nos sirve para funciones de *routing*, pasada una lista de identificadores de los nodos o una lista de direcciones MAC de *Bluetooth* como parámetro, nos devuelve según la interfaz en curso y el tipo de nodo, una lista de *cadena de caracteres* con las direcciones válidas de comunicación, es decir una lista de direcciones IP o una lista de direcciones MAC de *Bluetooth* de los siguientes nodos en la cadena de envío.

[1] En el momento de la redacción de este proyecto se ha descubierto que la función *deepHashCode()* tiene en cuenta los problemas citados y los resuelve. (ver `http://stackoverflow.com/questions/4671858/deephashcode-with-byte-array`)

- *send()*: recoge todos los parámetros y llama al método *write()* del controlador oportuno.

Los métodos *write()* de ambos controladores vistos bajo el modelo de caja negra son idénticos (salvo el parámetro lista de direcciones que toman como parámetro, pero para ello tenemos los métodos uniformadores que hacen que el trato desde el servicio a cada una de los controladores sea el mismo). Sin embargo la implementación difiere ya que no pueden producir los mismos resultados de la misma manera debido a que en *Bluetooth* la conexión está establecida y en *WiFi* hay que establecerla.

En *Bluetooth* se obtiene cada una de las hebras utilizadas en la conexión y se vuelcan los datos directamente al *socket* dónde se produce una escritura asíncrona que hace que ésta parezca inmediata. Realizamos lo mismo con todos los *sockets* donde haya que transmitir información y acto seguido confeccionamos un mensaje del tipo `MESSAGE_WRITE` para que sea manejado en el servicio, por último termina la ejecución del método *write()* sin devolver nada.

Código invulcrado

WifiController: write(), **SendoIP** (ServiceIntent), **sentReceiver** (nested BroadcastReceiver)

En *WiFi* sin embargo, el bucle de establecimiento y volcado de datos al *socket* nos haría esperar demasiado, pudiendo retrasar otras partes del flujo del programa causando problemas. En este caso dónde tenemos que levantar la conexión nos ayudamos de otra hebra en paralelo que haga esta misión (*serviceIntent*) y recogemos su resultado (cuando esté) gracias a un *broadcastReceiver* hubicado en el propio controlador. La acción que captura esta respuesta es `SendoIP.ACTION_SENT` y en ella encapsulamos:

- Un valor booleano que indique si la escritura ha sido correcta o no.
- Una *cadena de caracteres* que representa la dirección IP dónde se ha escrito.
- Un valor booleano que indique si se trata del último envío del mensaje de datos.

Cuando detectamos el final de un envío del mensaje en todos los nodos a los que va digirido, cuando devolvemos una respuesta igual que la de *Bluetooth* uniformando de nuevo el flujo y ocultando detalles de implementación al servicio, reforzando nuestro modelo de capas. Para ello hemos debido de almacenar al principio del proceso de envío los datos a enviar para poder devolverselos al servicio, este almacenaje se realiza sobre un objeto *Queue<byte[]>*, así evitamos que en el anuncio capturado por el *broadcastReceiver* incluir el mensaje a escribir en la red, lo que hace que la respuesta sea más ligera.

Por tanto el método *write()* del controlador *WiFi* al igual que su homólogo en *Bluetooth* no devuelve nada pero termina en tiempos parecidos, ya que en uno se producen escrituras asíncronas y en el otro inicializaciones de *serviceIntent* de la clase `es.upm.die.lsi.pfc.CWSNoA.util.SendoIP`

2.2.6.3. Forward de mensajes

Código invulcrado

Service: mHandler<controlador>, incomingContentCoordinator(), outgoingContentProgressCoordinator()

Al ser una mezcla de una recepción y un envío de datos, los detalles de implementación han sido relatados ya previamente, lo único que merece mención es la elección de las *cadenas de caracteres* que tras la finalización del flujo de recepción, son enviadas a través del flujo de envío. Esto hace que a efectos del servicio tenga el mismo punto de partida que un envío normal pero al detectarse las *cadenas de caracteres* especiales, el flujo se adapte y tenga en cuenta si ha de enviar mensajes de confirmación al nodo que produjo el primer envío (la recepción con la que empieza el proceso de reenvío).

2.2.7. Sensado del entorno compartido y política cognitiva

2.2.7.1. Parámetro del entorno: RSSI

Código invulcrado

Service: normalCognitiveTask(), mWifiAdapter.getConnectionInfo()

Gracias al adaptador *WiFi* (`android.net.wifi.WifiManager`) que nos facilita la plataforma *Android* podemos obtener un objeto de la clase `android.net.wifi.WifiInfo` a través del método *getConnectionInfo()*. El objeto obtenido nos facilita entre otros el valor del RSSI: *getRssi()*.

Como vimos en diseño, este valor también se puede obtener mediante los anuncios que envía el sistema *Android*, para ello registramos nuestro *broadcastReceiver* empleado para *WiFi*: `es.upm.die.lsi.pfc.CWSNoA.StatusReceiver_wifi`, a la acción `RSSI_CHANGED_ACTION`.

2.2.7.2. Parámetro del entorno: Intervalo promedio envío de mensajes

Código invulcrado

Service: normalCognitiveTask()
DataMessageQueue: getAvgArrivalRateUpdateTillNow()

Para calcular el promedio móvil almacenamos los valores a promediar en una *LinkedList*. Los valores a almacenar son de tipo *double*, representa el tiempo transcurrido entre mensajes entragados por las aplicaciones. Este objeto se hubica en la clase `es.upm.die.lsi.pfc.CWSNoA.util.MovingAverage` que es utilizada por la cola de mensajes cada vez que encola un mensaje. En la instanciación se elige el número de muestras de las que se va a tomar el promedio, en nuestro caso el tamaño de ventana es seis.
La razón de poner el promedio en una clase independiente es por si se necesita usar otro tipo de promedios (que puedan ser codificados como *double*) en otra parte de la arquitectura, por ejemplo su uso en políticas cognitivas.

2.2.7.3. Política cognitiva

Código invulcrado

Service: mayTakeAdvantageOfBT() mayTakeAdvantageOfWiFi(), sendTroublingRssi(), coordinatorCognitiveTask()

El envío por parte de datos del entorno con prioridad `URGENT` se realiza gracias al método *sendTroublingRssi()*, al recibirlo el coordinador vuelve a evaluar estos datos para evitar cambios de contexto disparados por umbrales que no concuerden entre los dos nodos.

Todos los parámetros que intervienen en la política cognitiva son modificables a través del archivo de prefencias que son:

Umbral WiFi Mínima intensidad de señal *WiFi* permitida medido en dBm

Zona de peligro WiFi acotación entre el umbral y un valor porcentual a éste.

Maximo decaimiento en zona de peligro Máxima pérdida de señal permitida en la zona de peligro.

Intervalo tiempo mínimo entre mensajes Mínimo tiempo aceptado entre envío de mensajes a través de la red medido en segundos

Zona de peligro cercano al intervalo acotación entre el intervalo mínimo y un valor porcentual a éste.

Máximo decrecimiento del intervalo en zona de peligro Si en la zona de peligro el intervalo disminuye porcentualmente al umbral en más de este valor procedemos a un cambio de contexto.

De WiFi a Bluetooth Está motivada por la pérdida de señal en la red *WiFi*. La decisión se basa en un umbral y en una zona de peligro. Al evaluar los datos de sensado de cierto nodo, vemos si el último valor de la RSSI está por debajo del umbral. En caso afirmativo disparamos el proceso de cambio de contexto o *handover*, si no evaluamos si el valor promeadiado a un número de muestras elegible si este valor está en la zona de peligro vemos la evolución de la señal, si decae en más de lo permitido procedemos al cambio de contexto.

De Bluetooth a WiFi Está motivada por el aumento de mensajes en la red. La decisión se basa en un umbral y en una zona de peligro. Si el último dato de sensado de cierto nodo se ve que el tiempo (ya promediado de manera móvil) entre mensajes está por debajo del umbral se producirá a el cambio de contexto o *handover*. En caso contrario vemos si la media de los últimos datos sensados está en la zona de peligro, si es así, vemos si el último valor se aleja más de lo permitido.

2.2.8. Cambios de contexto

Los cambios de contexto o *handover* usan protolocos, mensajes de red y otros procesos ya descritos tanto en este capítulo como en diseño.

Mencionar que para ayudar en la tarea de uniformidad y que procesos parecidos empiecen en el mismo punto de partida, parte del flujo del cambio de contexto como implica configurar una interfaz, se reutiliza la misma *AsyncTask* utilizada para el levantamiento de la interfaz con pequeños pasos intermedios que antes no se ejecutaban y ahora sí, para ello debemos pasar como parámetro si estamos configurando la interfaz para un levantamiento normal o como consecuencia de un cambio de contexto.

Los *timeouts* utilizados para evitar *deadlocks* se implemtan siguendo el esquema de programación basado en *Runnables* siguiendo las directrices de la interfaz `java.util.concurrent.ScheduledFuture<V>` y ejecutados en hebras gestionadas por `java.util.concurrent.Executor`

Al programar una tarea (el objeto *Runnable*) con el objeto *Executor* a través de su método *schedule()* obtenemos una referencia a la interfaz *Future*, esta interfaz nos permite consultar si una tarea ha sido o no realizada y la cancelación de ésta.

2.3. Aplicación utilitaria servicio cognitivo

La aplicación desarrollada se apoya en dos componentes *Android*: *Activity* [`App_Activity`] y *PreferenceActivity* [`App_Settings_Activity`], siendo la clase `es.upm.die.lsi.pfc.CWSNoA.App_Activity` la que vertebra toda la aplicación.

Esta clase extiende del componente `FragmentActivity` para permitir la compatibilidad hacia atrás de ciertas funciones (hasta la versión 1.6 de *Android*) haciendo uso de la libreria *Support Library* [12]. A efectos prácticos actúa como cualquier *Activity*, este componente hace de contenedor y lanza a otros elementos, centralizando la comunicación con la 'capa cognitiva', los componentes que contiene son *Fragments* [13], un fragmento es un trozo autónomo contenido en una *activity*. Aunque pueden no tener interfaz gráfica, su uso está pensado para ocupar un lugar que junto a otros fragmentos construyan la interfaz de usuario. Esta construcción puede ser dinámica y reusable ya que cada trozo puede ser reutilizado en cualquier sitio. Las comunicaciones entre fragmentos se realizan a través de la *activity* que los contiene. La *activity* por ser dueña del fragmento tiene acceso a todos los métodos de éste, además los eventos producidos en un fragmentos (como la pulsación de un botón) pueden ser capturados directamente en la propia *activity*. Por lo que la comunicación hacia abajo (*activity* → *fragment*) es bastante sencilla al basarse en invocación.
Sin embargo la comunicación hacia arriba (proveniente desde el *fragment* se basa en la declaración de interfaces de métodos. Estos métodos tienen que ser implementados en la *activity*, el fragmento puede forzar a que la *activity* que le contenga cumpla esa interfaz de métodos, de tal forma, que se asegure la ejecución mediante la invocación del método en la *activity* desde la referencia obtenida en el método *onAttach()* en el fragmento, haciendo a éste autónomo en el paso de información. La comunicación entre fragmentos se basa en una mezcla de los dos procedimientos, una comunicación hacia arriba y una vez se está en el *scope* de la *activity* realizar una comunicación hacia abajo al fragmento deseado.
En la figura 2.5 se ve la comunicación entre fragmentos con flecha discontinua debido a que la *activity* media en este intercambio de información.

En la figura vemos los componentes que dan sentido a las tres secciones descritas en diseño, las dos primeras están albergadas en la `App_activity` que se compone de:

- SERVICE CONTROLLER Es un fragmento que no tiene interfaz gráfica y que marcamos con la sentencia *setRetainInstance(true);* como 'no destruible' frente a cambios en la configuración (como puede ser un volteo de la pantalla, o un cambio del idioma...) para no perder el vínculo con el servicio cognitivo. En él, se implementan los procesos de *bind* y todas las comunicaciones de la API. Este fragmento

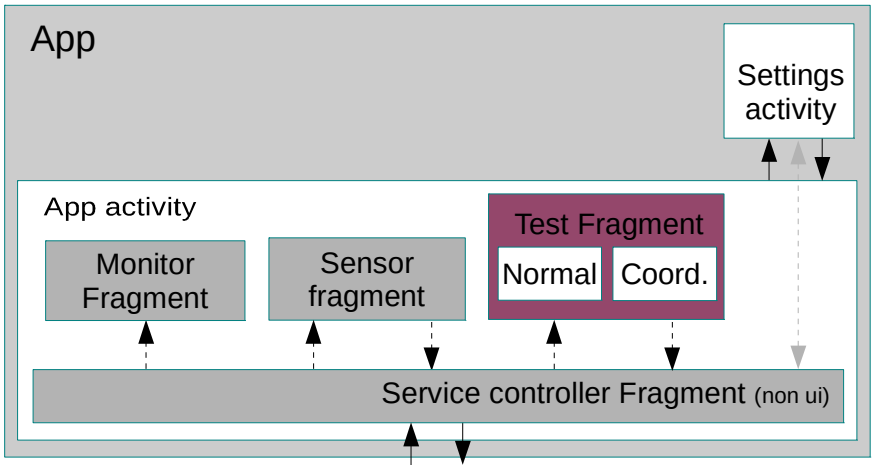

Figura 2.5: Diagrama de bloques de las secciones que componen la aplicación y su comunicación entre ellas

ubicado en la clase `es.upm.die.lsi.pfc.CWSNoA.App_serviceController_Fragment` da soporte a los demás componentes a través de la *activity* que lo contiene, convirtiéndose en la puerta de entrada a la 'capa cognitiva'.

- MONITOR FRAGMENT Monitoriza el estado del servicio, mostrando el estado de la conexión, la interfaz utilizada y el nombre del coordinador. La comunicación es unidirecional entre la clase `es.upm.die.lsi.pfc.CWSNoA.App_monitor_Fragment` y el fragmento que controla la comunicación con el servicio

- SENSOR FRAGMENT Ubicamos en `es.upm.die.lsi.pfc.CWSNoA.App_sensor_Fragment` la simulación de un sensor. El fragmento extiende la clase `ListFragment` para presentar y manejar en pantalla un historial de mensajes recibidos y enviados. Junto a ello se presenta un formulario para introducir y enviar el valor del sensor. Necesitamos de una conexión bidirecional para en un sentido enviar el dato y en el otro recibir y actualizar la lista de nodos que se muestra.

- TEST FRAGMENT Este fragmento implementa por si sólo (en conjunción con el fragmento `App_serviceController_Fragment`) la segunda sección, mientras que los arriba enumerados componen la primera sección focalizada en el función de la aplicación. Dentro de este fragmento: `es.upm.die.lsi.pfc.CWSNoA.App_testing_fragment` se ubican otros dos que son cargados dinámicamente según el tipo de nodo que se trate:

  - `es.upm.die.lsi.pfc.CWSNoA.App_testing_normal_Fragment` provee una manera al usuario de guardar un valor en el servicio que será lo que envíe al recibir una petición de `TEST`.
  - `es.upm.die.lsi.pfc.CWSNoA.App_testing_coordinator_Fragment` en caso de nodo coordinador permite selecionar un nodo gracias a un *spinner* modificado para que la acción de pulsar devuelva siempre un valor aunque sea el mismo que tenía selecionado. Al interactuar con él se lanzan los procesos de red necesarios para obtener el valor almacenado en el servicio del nodo remoto.

Un detalle de implementación que cabe destacar de esta sección de test es el envío periódico de datos del sensor. En vez de arrancar objetos pesados como una hebra o jugar con un *timer* se ha decidido trabajar como trabaja *Android* que es mediante el envío de mensajes a colas del sistema, para ello:

1. Creamos un objeto *Handler* para obtener una cola y manejar mensajes.
2. Con el método *postDelayed()* mandamos un mensaje a la cola pasados los milisegundos que se indican como parámetro, el otro parámetro necesario es un objeto *Runnable* donde definimos nuestra funcionalidad. Al final del método *run()* mandamos otro mensaje a la cola pasándole como objeto *Runnable* este mismo y el mismo retraso.
3. Al transcurrir el periodo especificado el mensaje llega a la cola del sistema y se ejecuta el objeto *Runnable* quien deja preparado otro mensaje en la cola para su posterior ejecución, mediante una variable de control podemos dejar de progrmar el envío de mensajes a la cola.

Este esquema de programación, no introduce *sleeps* ni bloqueos, se gestiona con un procedimiento nativo de *Android* lo que permite ser eficientes al no consumir casi recursos.

La tercera sección se hubica en su totalidad en la clase `es.upm.die.lsi.pfc.CWSNoA.App_Settings_Activity` es un componente especializado para hubicar preferencias ya que muestra un *layout* conveniente para mostrar una serie de valores con su descripción y maneja el archivo de preferencias donde se hubican éstas.
Esta sección es lanzada por la `App_Activity` a través del menú. Se lanza con un *intent* que espera resultado, es decir cuando el usuario retorne de la configuración, *Android* informa al que lanzó el intent con un código de resultado:

Constante	Valor	Máscara
`Activity.RESULT_CANCELED`	(definido por la plataforma)	
`RESULT_UPDATE_APP_STUFF`	'1'	1
`RESULT_UPDATE_SERVICE_STUFF`	'2'	10
`RESULT_CHANGES_MISC`	'4'	100
`RESULT_CHANGES_TEST`	'8'	1000

En `App_settingsActivity` vamos, de acuerdo a estas máscaras, modificando el valor a devolver, no se produce enmascaramiento del valor debido a que las máscaras son ortogonales entre si. Al recibirlo en `App_activity`, vamos pasando este valor por cada una de las máscaras descubriendo si ha habido algún cambio en sección: parámetros cognitivos (aplicación, servicio), preferencias o test. Al acabar todos los filtros los cambios ocurridos son evaluados y agrupados si hemos de traspasarlo al servicio. Otros cambios son a nivel aplicación como la carga dinámica de la sección de test o el borrado automático del valor del sensor al enviar el dato.

Estas máscaras permiten que si el usuario no hace nada (se devuelve un valor '0' que corresponde a `Activity.RESULT_CANCELED`) no informar al servicio y si ha habido cambios no tener que revisar toda la configuración para ver que ha cambiado y actuar en consecuencia, el valor devuelto nos indica dónde se ha producido el cambio.

Referencias/Bibliografía

[1] Rabaey, J.; Wolisz, A.; Ozer Ercan, A.; Araujo, A.; Burghardt, F.; Mustafa, S.; Parsa, A.; Pollin, S.; I-Hsiang Wang; Malagon P., "Connectivity Brokerage - Enabling Seamless Cooperation in Wireless Networks," tech. rep., Berkeley Wireless Research Center, Sep 2010.

[2] Araujo, A.; Romero, E.; Blesa, J.; Nieto-Taladriz, O., "A framework for the design, development and evaluation of cognitive wireless sensor networks," *International Journal On Advances in Telecommunications*, vol. 5, pp. 141 – 152, Dic 2012.

[3] Wikipedia, "Signalling System No. 7." `http://en.wikipedia.org/wiki/Signalling_System_No._7`. Último acceso el 17-9-2014.

[4] "Developer guides: Scheduling Repeating Alarms." `https://developer.android.com/training/scheduling/alarms.html`. Último acceso el 17-9-2014.

[5] Mark L. Murphy, *The Busy Coder's Guide to Advanced Android Development*. CommonsWare, 2011.

[6] "Google developers: Protocol Buffers." `https://developers.google.com/protocol-buffers/`. Último acceso el 17-9-2014.

[7] "Developer guides: AsynTask." `http://developer.android.com/reference/android/os/AsyncTask.html`. Último acceso el 4-9-2014.

[8] "Communications Topology: Piconet Topology." `https://developer.bluetooth.org/TechnologyOverview/Pages/Topology.aspx`. Último acceso el 4-9-2014.

[9] "RFCOMM: Bluetooth Development Portal." `https://developer.bluetooth.org/TechnologyOverview/Pages/RFCOMM.aspx`. Último acceso el 4-9-2014.

[10] "Developer guides: Preferences." `http://developer.android.com/reference/android/preference/Preference.html`. Último acceso el 4-9-2014.

[11] "Java's hashCode is not safe for distributed systems." `http://martin.kleppmann.com/2012/06/18/java-hashcode-unsafe-for-distributed-systems.html`. Último acceso el 4-9-2014.

[12] "Developer guides: Support Library." `http://developer.android.com/tools/support-library/index.html`. Último acceso el 17-9-2014.

[13] “Developer guides: Fragments.” http://developer.android.com/guide/components/fragments.html. Último acceso el 24-9-2014.

Lista de Acrónimos

ACK ..... *Acknowledgment*

ANR ..... *Android Not Responding*

AP ..... *Access point*

API ..... *Application Programming Interface*

CB ..... *Connectivity Brokerage*

CR ..... *Cognitive Radio*

CWSN ..... *Cognitive Wireless Sensor Network*

FIFO ..... *First-in First-out*

IDL ..... *Interface Definition language*

IP ..... *Internet Protocol*

MAC ..... *Medium Access Control*

PPP ..... *Point-to-point*

RSSI ..... *Received Signal Strength Indication*

SQL ..... *Structured Query Language*

SS7 ..... *Signalling System no. 7*

SSID ..... *Service set identification*

TCP ..... *Transmission Control Protocol*

UDP ..... *User Datagram Protocol*

UUID ..... *Universally unique identifier*