Capítulo 1

Diseño

Contar qué es una arquitectura modular y comparación con una arquitectura cognitiva.
vamos a citar cosar [1] acronimo! CWSN

1.1. Arquitectura cognitiva

Explicación de los diferentes bloques de una arquitectura cognitiva: Repositorio, ejecutor, optimizador....

1.2. Arquitectura implementada

Características de nuestra arquitectura: modular, homogénea, escalable (hasta 7 nodos) lo que le hace especialmente compatible a cambios en el futuro ya que los procesos están fuertemente marcados e independientes.

1.2.1. Comparación ambas arquitecturas

Clasificar nuestros metodos en los bloques de una arquitectura cognitiva.

1.3. Detalles de implementación

Servicio vs activity.... multi app...

1.4. Cómo montar una aplicación sobre el servicio cognitivo, definición de la API

La forma de interactuar con el servicio cognitivo se basa en, como se ha visto en 1.3, 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 se monte por tanto 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. Es necesario su almacenaje ya que es necesaria para la confección de algunos mensajes de la API y a que no hay una manera activa de pedir la lista de nodos 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
OUTGOING_CONTENT_RESULT	6
INCOMING_CONTENT_PROGRESS	7
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 1.1: 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.

1.4.1. 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. 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 1.2):

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 1.2: 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 de este mensaje (ver tabla 1.3):

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 1.3: Mensaje entrega parámetros de aplicación

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 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 es (ver tabla 1.4):

La información de cada campo:

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

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 1.4: Mensaje finalización de *handshaking*: CWSN no establecida

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

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 1.5.2)

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 1.5):

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 1.5: Mensaje finalización de *handshaking*: CWSN establecida

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"

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.

1.4.2. 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 1.6):

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 1.6: Mensaje para el envío de datos

1.4.3. 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 1.7)

La información que podemos leer de este mensaje es:

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:

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 1.7: Mensaje resultado envío de datos presentado a la aplicación

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.

1.4.4. 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 1.8):

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 1.8: Mensaje datos recibidos

1.4.5. 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 plasma todos los cambios de estado que haya a lo largo de un proceso completo. En otros mensajes enviados por el servicio que aportan parte de esta información, ésta corresponde tan sólo 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 1.9: Estados de la interfaz de comunicación

Los eventos se producen entre dos nodos con contacto directo, es decir, 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 1.5) que para los controladores son tratados de la misma manera que otros mensajes de red.

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_CANT_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
EVENT_INTERFACE_END_SWITCHING	8
EVENT_UPDATE_LIST_NODES	9

Tabla 1.10: 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 1.11):

El detalle de cada uno de los campos es:

Arg1: Interfaz Un entero cuyos valores corresponden a:

0. *Bluetooth*

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 a los cuales no les ha llegado el mensaje
nodeNameList	Un array de *cadena de caracteres* con los nombres de los nodos a los cuales no les ha llegado el mensaje

Tabla 1.11: Mensaje información evento/estado por parte del servicio

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

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

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 1.9, 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 1.10, 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_CANT_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.

1.4.6. 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 reliza 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 estas. Algunos de los errores de los que son manejados por el servicio se listan en la tabla 1.12.

Que son transmitidos a la aplicación o las apliaciones por medio de este mensaje (ver tabla 1.13):

1.4.7. Mensajes encolados

Puede darse la circustancia de que al realizar el envío de datos al servicio, éste no puede enviarlo por la red ya que se encontra en un cambio de contexto. En estas ocasiones los mensajes recibidos por parte de las aplicaciones son encolados.

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 1.12: Mensajes de error por el no cumplimiento de la API

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 1.13: Mensaje de error recibido del servicio

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 enviado el mensaje, ya que puede haber cambios 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 1.14):

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 1.14: Mensaje 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 estamos intentando enviar el mensaje y los dos últimos están rellenos de los nodos que al no encontrarse ahora 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 1.4.3

1.4.8. 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 1.4.1). Empezaremos enviando directamente el segundo mensaje, con nuestos nuevos parámetros al servicio con el mensaje ya visto en la tabla 1.3. 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.

1.4.9. 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 sola 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:

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 1.15: Mensaje desregistro cliente

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

1.4.10. 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 1.16):

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 1.16: 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 enviamos 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.

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

Actualización de parámetros avanzada

Un nodo cognitivo tiene mas parámetros (como se puede ver en 1.5.5) 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 1.17). Los pormenores de este flujo los veremos en la sección 1.5

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

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 1.17: Actualización de todos los parámetros del servicio

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 sobre el 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 respues a enviar en el campo *obj* del mensaje (ver tabla 1.18):

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 1.18: Mensaje *workaround* salvado de respuesta

El otro agente, el nodo coordinador, enviando este mensaje al servicio, recupera la información del otro actor (ver tabla 1.19):

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 1.19: Mensaje *workaround* petición de respuesta

Una vez enviada esta petición la respuesta nos vendrá encapsulada en un mensaje del mismo tipo (ver tabla 1.20). 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.

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

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 1.20: Mensaje *workaround* respuesta obtenida

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 1.21: Mensaje *workaround* cambio de interfaz de comunicación

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 1.22):

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 1.22: Mensaje *workaround* encolamiento de mensaje de datos

Y para vaciar la cola de mensajes en cualquier momento deberemos enviar este otro (ver tabla 1.23):

1.5. 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 1.4 ya que dónde empieza o finaliza la acción es con mensajes intercambiados de la API.

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 1.23: Mensaje *workaround* vaciado cola de mensajes

1.5.1. Registro de una aplicación en el servicio cognitivo

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 1.4), 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 1.2

El siguiente mensaje (ver tabla 1.3) 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 1.4.8 o 1.5.5. Con esta información el servicio dependiendo del punto en que se encuentre actuará de una forma u otra.

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 el servicio pero no hay conectividad con otros nodos, en otras palabras, estamos solos en la red.

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 ya vista en las listas de la sección 1.4.1. Puede verse el detalle de este mensaje en la tabla 1.4

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 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 1.27) y el estado de la interfaz. La estructura del mensaje devuelto puede apreciarse en la tabla 1.5).

1.5.2. 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, esta configuración se realiza en una *AsynTask* de *Android* que permite descargar la *UI-Thread* evitando errores del tipo *ANR "Application Not Responding"* ya que parte de la tarea se ejecuta en segundo plano, además este tipo de tarea habilita mecanismos de paso de información entre ambas hebras. Antes de configurar una interfaz pediremos a los controladores de las otras interfaces que liberen los recursos (llamando a su método *stop*) para poder partir de una situación deseable.

WiFi: servicio y controlador

Service: SetupWifiInterface (asynTask)
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, lo primero que hacemos es intentar parar las hebras asociadas a la interfaz *WiFi* que cerrarán los *serverSockets* y el *broadcastReceiver* útil para registrarse a eventos de información relativa a la interfaz *WiFi* como son RSSI y los eventos de conexión/desconexión a un punto de acceso. Partiendo de una situación en que todos los recursos están liberados, encendemos la interfaz si no lo estuviese ya, esperamos a que se encienda mediante una espera de consulta a una variable que es modificada por *StatusReceiver_ wifi.java*, nuestro *broadcastReceiver* para la interfaz *WiFi*. Si nos han pasado como parámetro el SSID, intentamos conectarnos a él desconectándonos del punto de acceso acutual si no fuese el mismo. 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 arrancando las hebras del controlador dependiendo del tipo de nodo que seamos:

Normal Arrancamos una hebra que escuche gracias a un *serverSocket* para que acepte conexiones TCP, una vez estemos escuchando en el puerto indicado, enviamos el mensaje de tipo REGISTER_EXCHANGE con el campo EXTRA "errorSetupInterface" a '*false*' y el resto de campos descritos en la tabla 1.4 y lanzamos el proceso de registro en la red.

Coordinador Arrancamos un par de hebras, una para tráfico TCP que admita conexiones fiables y otra para el tráfico UDP que escuche y capture paquetes, si todo se ha configurado bien enviamos un mensaje análogo al párrafo anterior y quedamos a la espera de capturar paquetes.

Bluetooth: servicio y controlador

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

Como en el controlador de *WiFi* empezamos (en segundo plano) liberando si no estuviesen ya cada uno de los recursos, en *Bluetooth* coexisten dependiendo del tipo de nodo 3 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. Encendemos la interfaz *Bluetooth* si no estuviese encendida ya, mediante una petición al sistema *Android* y esperando mediante *polling* a que *StatusReceiver_bt.java*, el *broadcastReceiver* para la interfaz *Bluetooth* modifique la variable de control de esta espera. Una vez finalizada, procedemos a configurar las hebras del controlador según el tipo de nodo.

Normal Este tipo de nodo se comporta como esclavo en la comunicación *Bluetooth* por lo que necesita arrancar una hebra para aceptar conexiones, como sólo esperamos un maestro cuando consigue una conexión (disparada sobre el mismo UUID por el que hemos registrado el socket) liberamos este recurso. Como ya hemos configurado la interfaz enviamos el mensaje final del *handshaking* (ver tabla 1.4) con las instrucciones que hemos visto ya.

Coordinador Este tipo de nodo se comporta 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() que se ejecuta en su propia hebra para asegurarnos que no se ejecuta en la hebra de la interfaz de usuario, en este caso no hace falta ya que la estamos llamando desde el segundo plano, pero como veremos este método se llama también directamente desde la UI-thread. Este método coge como parámetros la dirección MAC con la cual queremos establecer una conexión y el número de intentos que vamos a intentar establecer esta conexión (a parte de otras constantes como el UIID), como en *Bluetooth* es fácil que falle el establecimiento intentamos minimizar este riesgo reintentando la conexión. En el momento que conseguimos establecer la conexión, liberamos esta hebra que se volverá a crear con el siguiente intento de conexión a otro nodo y manejamos la conexión con una nueva hebra del tipo *connectedThread*, al final tendremos tantas hebras como nodos activos en la red. La forma en la que sabemos cuando podemos pasar al siguiente nodo radica en 2 flags que son modificadas gracias al sistema de estados: setState(), que tienen los controladores. En esencia, cuando disparamos una conexión y ésta falla propagamos un estado de conexión fallida al servicio que aumentara el flag de conexiones fallidas (*numFailConnection*), si la conexión se produce con éxito se propaga análogamente un estado de nuevo nodo en la red y modificaremos el flag de nuevo dispositivo (*flagNewDevice*) saliendo de la espera en la que nos encontramos, por establecimiento de la conexión o por llegar al máximo de intentos.

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 1.4.

1.5.3. 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.

WiFi

WifiController: SendHelloPacket (AsynTask), 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 otra *AsynTask* que comienza a enviar paquetes cada 800 milisegundos hasta llegar a un máximo de 20. La razón de enviar una ráfaga de paquetes radica en que al ser trafico broadcast/multicast los puntos de acceso pueden menospreciarlo (incluso bloquearlo) y 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 de 20 paquetes espaciados 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 el éste quien informará a todas las aplicaciones registradas del fallo del proceso indicando que no ha sido posible establecer una CWSN (EVENT_INTERFACE_CANT_CONNECT).

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 1.24: 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, así evitamos propagar información al servicio de paquetes que no encajen con la estructura y campos requeridos desechando otros paquetes. Si capturamos un *"HelloPacket"* generaremos un evento 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 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 es:

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

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 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 1.4.5

[1] Notar que el campo 'Dirección IP' puede no ser informado porque no este disponible en ese momento o sea inecesario

con el evento EVENT_INTERFACE_NEW_NODE (visto en 1.10) y anunciará la lista de nodos quién es el coordinador.

Bluetooth

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 esta conexión, es decir al crear una hebra del tipo *ConnectedThread*, 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 MAC para obtener su identificador y pospondríamos el anuncio del nuevo nodo hacia las aplicaciones hasta que no obtengamos el mensaje de señalización donde 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 enviamos un mensaje de petición de información para que nos informe de su nombre y papel, ya que su 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 1.11

1.5.4. Salida de un nodo de la 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 1.15). 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* lo mismo pasa con las peticiones de *start* y *stop*, teniendo 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, el flujo que provee *Android* ha sido modificado para que cuando el servicio deje de tener aplicaciones sobre él, el mismo se destruya.

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 volunaria 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 (incluidas en el método *onDestroy()* del servicio) 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 hacer estas acciones para desembocar en la misma situación inicial.

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

WiFi

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 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, solo podremos mandarlo sobre TCP ya que los nodos normales no escuchan sobre UDP como hemos visto en 1.5.2. Este envío se realiza gracias a otra asynTask que al finalizar libera los recursos del controlador cerrando la vida de la aplicación por completo.

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 1.26: Campos del paquete "ByePacket"

Normal Al recibir de manera fiable el mensaje 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.

Coordinador Al recibir el mensaje 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.

Bluetooth

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*, las hebras que haya abiertas desaparecen lo que acarrea que el otro extremo de la comunicación se de cuenta de que el *socket* usado en la comunicación ha sido cerrado, lo que supone que el controlador detecte el cierre de la conexión y lance el evento de pérdida de nodo.

Normal Si somos un nodo normal, 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.

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 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.

1.5.5. Actualización de parámetros en la red

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.

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 = secundario mientras que 1 = primario
Código de aplicación	Una *cadena de caracteres* para discernir entre mensajes enviados por distinas aplicaciones

Tabla 1.27: 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 sus 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 mas restringida. En otras palabras si una aplicación pide ser "nodo primario" y otra aplicación pide ser "nodo secundario" el valor del parámetro será "nodo primario".

La primera forma de actualizar los parámetros de aplicación es con el registro de la aplicación en el servicio (ver sección 1.5.1) o en cualquier momento enviando un mensaje desde la aplicación al servicio del tipo (REGISTER_EXCHANGE) (visto en la tabla 1.3) 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 1.17.

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, no nos preguntará sino que saca una copia de su base de datos los parámetros, que son los mismos que tendrá el resto de la red, pues son la copia desactualizada de la base de datos del coordinador y no de los nuestros. 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*.

1.5.6. 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.

Recepción de mensajes

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

Empezamos explicando este proceso por ser el más fácil. Cuando la hebra del controlador (no importando cual) sale de su bloqueo al recibir datos, se genera el evento de recepción de datos que es manejado en el servicio. 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 1.28).

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 29). 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 1.29) 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

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 1.28: Mensaje recibido por la red entregado por ambos controladores al servicio

recuperado la aplicación destinataria y el nodo que envío el mensaje, podemos trasladar esta información a la aplicación correspondiente (ver tabla 1.8).

Envío de mensajes

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 1.6)

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 este es el tipo de dato que se espera en el campo *obj* del mensaje (*payload*), si no el servicio devolvera 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 1.29.

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 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

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 1.29: Campos del mensaje de datos

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, esto depende del tipo de nodo y del mapa de red. 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. 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)

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 7 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 nuestro caso 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 1.30).

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 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 1.30: 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 1.7).

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 vista desde un nodo normal. En este caso al recibir por parte del controlador el mensaje (ver tabla 1.30) 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 1.31.

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 1.7)

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

Detalle implementación envío en WiFi Un comentario más acerca de la implementación de cómo enviamos el mensaje dependiendo de la interfaz, como se ha visto ya, enviar

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 1.31: Campos del mensaje ACK

consiste en llamar al método *write()* del controlador correspondiente y pasarle los datos adecuados. En el caso de *Bluetooth* como la conexión o conexiones ya están establecidas, podemos permitirnos recuperar la hebra de conexión deseada y volcar los datos, esperando a devolver una respuesta en la misma petición.

Sin embago en el caso de *WiFi* esto 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. Por tanto el método *write()* del controlador *WiFi* no devuelve ningún resultado a su llamada a diferencia de su homólogo en *Bluetooth* y es cuando detectamos que hemos terminado de enviar todos los mensajes correspondientes a un envío de un mensaje de datos cuando devolvemos una respuesta igual que haciamos en *Bluetooth* uniformando de nuevo el flujo y ocultando detalles de implementación al servicio, reforanzo nuestro modelo de capas.

Forward de mensajes

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

Esta sección es una mezcla de una recepción y un envío de mensajes, reservada al papel de coordinador. Este proceso comienza con la recepción 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 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), 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. No se puede utilizar el "Código de aplicación" grabado en el mensaje de datos ya que puede haber dos o más aplicaciones registradas con el mismo código. El parámetro por tanto es una simple *cadena de caracteres* que representa el *messenger* de la aplicación, que gracias a una serie de *HashMap* nos relaciona ambos datos. De esta manera evitamos por tanto serializar un objeto complejo o ubicarlo en un sitio que no está pensado para ello, debido a que al estar en capas diferentes el campo (*msg.replyTo*) se usaría con otro propósito 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.

Cuando el mensaje requiere de un re-envío, no existe una aplicación para rellenar este parámetro, elegimos dos *cadena de caracteres* reservadas que son "forward-me" y "forward-not-me" para referirse a la situación en la que el coordinador forma parte de la lista del mensaje que se reenvía y cuando no. Nuestro flujo al reconocer estas *cadena de caracteres* se da cuenta que se trata de un reenvío.

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 1.30 pero es aquí dónde en vez de trasladar el mensaje a la aplicación (dado que no existe) es dónde confeccionamos el mensaje de confirmación (ver tabla 1.31) que enviaremos al originario del mensaje de datos, en vez de informar a la aplicación como se hace en el envío.

1.5.7. Sensado del entorno

Los parámetros del entorno que son de nuestro interés son el *Received Signal Strength Indication* (RSSI) 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 1.32) 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.

Parámetro del entorno: RSSI

Service: normalCognitiveTask(), mWifiAdapter.getConnectionInfo()

Para medir la intensidad de señal podemos utilizar de manera activa la API de *Android* que nos facilita a través del adaptador de *WiFi* (WifiManager) el objeto "WifiInfo", el método getRssi() nos devuelve 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, para ello suscribimos nuestro *broadcastReceiver* de *WiFi* a los mensajes de cambio del RSSI.

En nuestra arquitectura tenemos dos puntos de medida de este parámetro, el primero está localizado en la tarea cognitiva dónde activamente obtenemos este valor, que guardamos en la base de datos para usos futuros. El otro punto reside en las notificaciones sobre cambios del parámetro gracias a que estamos registrados a mensajes del tipo RSSI_CHANGED_ACTION, estas notificaciones nos ayudan a actuar con el menor tiempo de respuesta, ya que si detec-

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 1.32: Campos del mensaje SCAN

tamos una situación que entrañe riesgo podemos compartir esta información sin esperar a la tarea cognitiva donde realizamos entre otras cosas el sensado del entorno.

Parámetro del entorno: Intervalo promedio envío de mensajes

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 cooridnador, obtenemos el promedio y registramos de nuevo el tiempo así podemos re-calculamos 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).

1.5.8. 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.

Cambio de contexto con interfaz destino WiFi

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

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 1.5.2 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 1.33)

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 MAC del punto de acceso para evitar cualquier equívoco
  - **Dirección IP coordinador** Una *cadena de caracteres* con la dirección IP del nodo

Tabla 1.33: 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 1.5.2 y 1.5.3.

El coordinador al recibir los "HelloPackets" 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 1.34

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

Si antes de que salte el *timeout* programado, hemos recibido todos los "HelloPackets" 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.

Coordinador 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 1.35

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.

Normal Al recibir el mensaje de señalización de cambio de contexto, procedemos a cancelar el envío de "HelloPackets" por si no hubiesemos recibido el mensaje visto en la tabla 1.34, 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.

Cambio de contexto con interfaz destino Bluetooth

Service: coordinatorCognitiveTask(), mayTakeAdvantageOfBT(), contextChangeToBT(), SetupBTinterface (asyncTask), mHandlerBt, tellBTisUP(),

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 1.25). Tendremos tantas representaciones como nodos haya en la red.

Tabla 1.35: Campos del mensaje finalización cambio de contexto

scheduleEndSwitchBTRunnable (future) scheduleSetupBTRunnable (future) , sendEndSwitchMessage(), endContextualChange()

Cuando se produce un cambio de interfaz a *Bluetooth* el coordinador como antes 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 de contexto hacia *Bluetooth*, este mensaje tiene la disposición mostrada en la tabla 1.36

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 1.36: Campos del mensaje SWITCH hacia *Bluetooth*

Una vez enviado, programamos la tarea de configurar la interfaz de *Bluetooth* en el coordinador con un margen de tiempo suficiente para obtener las respuesta 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 1.34, 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. Procede por tanto a configurar la interfaz y disparar una ronda de conexiónes 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 1.35) 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 la confirmación al coordinador de que está listo, programa una tarea (implementada gracias al esquema de programación basado en *Runnables* alojados en la interfaz *Future* y ejecutados en hebras gestionadas por el objeto *Executor*) 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.

Referencias/Bibliografía

[1] Perera, C.; Zaslavsky, A.; Christen, P.; Georgakopoulos, D., "Context Aware Computing for The Internet of Things: A Survey," *Communications Surveys & Tutorials, IEEE*, vol. abs/1305.0982, pp. 1 – 4, May 2013.