

Capítulo 1

Diseño

Contar qué es una arquitectura modular y comparación con una arquitectura cognitiva.

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

Tipos de mensajes que se intercambian, y como tienen que ir rellenos estos.

1.5. Procesos de red

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 nos tenga en cuenta, el segundo para informarle de nuestros parámetros de aplicación y el tercero será una confirmación del servicio hacia la aplicación informando de todos los parámetros del servicio respecto a nuestra aplicación.

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. Tras esto 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 son:

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

Tabla 1.1: Mensaje registro cliente

En el siguiente mensaje que debemos enviar, informaremos acerca de nuestros parámetros de aplicación como es: 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.

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

Tabla 1.2: Mensaje intercambio parámetros aplicación

Este mensaje ha sido separado del anterior para poder reutilizarlo e introducirlo en el flujo descrito en 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 hay aplicaciones ya montadas sobre el servicio pero no hay conectividad con otros nodos, en otras palabras, estamos solos en la red. (Hay otra forma de llegar a esta situación que veremos en 1.5.5, que no explicamos ahora para no enmarañar el texto).

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 nos devolverá (a todas las aplicaciones montadas) información acerca de todos los parámetros y el estado actual de la red. A saber:

Estado de la interfaz Un entero que representa el estado de la interfaz 0 = Down, 1 = Idle (no en red), 2 = Idle (en red), 3 = conectando (estado visto sólo en *Bluetooth*), 4 = Enviando, 5 = Recibiendo.

Interfaz Un entero cuyo valor 0 representa a *Bluetooth*, el valor 1 representa a *WiFi* y -1 representa a una interfaz desconocida (útil para casos de error y cambios de contexto)

Papel del nodo Un entero cuyo valor 0 = secundario y 1 = primario

Tipo de nodo Un entero cuyo valor 0 = normal, 1 = coordinador y 2 = coordinador temporal en *Bluetooth* (no usado en estos momentos)

Periodo tarea cognitiva Un *double* que representa los segundos que transcurren entre ejecuciones de la tarea cognitiva

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

Resultado configuración interfaz Un valor booleano con el resultado de configurar la interfaz de comunicación de forma correcta

El detalle del mensaje, queda:

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

Tabla 1.3: Mensaje finalización de *handshaking*: CWSN no establecida

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 nos devolverá (en exclusiva) los parámetros del servicio (ver lista página 2) menos el resultado de configurar la interfaz ya que no ha sido necesaria ninguna configuración pero incluye estos nuevos:

Código de la aplicación Un *cadena de caracteres* que representa el código de la aplicación

Lista nodos TODO TODO TODO en el código

El detalle del mensaje queda ¹:

¹Notar que el campo del mensaje *obj* viene informado con el valor booleano *True* lo que nos ayuda a distinguir si previamente el servicio ya cooperaba en una red, a parte que esta respuesta es casi inmediata, a diferencia de la anterior.

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

Tabla 1.4: Mensaje finalización de *handshaking*: CWSN establecida

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 *AsyncTask* de *Android* que permite descargar la *UI-Thread* evitando errores del tipo *ANR* “*Application Not Responding*” ya 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 actual 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 1.3 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* 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 UUID), 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 1.3.

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

```

Tras la configuración de la interfaz por parte de un nodo normal, éste inicia otra *AsyncTask* 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 tráfico 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:

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 desconido)

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

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 al nodo es: Al recibir esta

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

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 comprobar 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 extraídos del mensajes de señalización. En este momento el nodo normal informará a las aplicaciones de que ha entrado en la red 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 configura 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.

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. El proceso en el lado del nodo que se va, termina con la preparación de la base de datos para una nueva instancia del servicio, estas acciones comprenden resetear las direcciones IP y el identificador del coordinador para cada nodo, así como el marcaje de éstos como inactivos, por último una indicación de que el servicio ha sido parado de forma voluntaria. De tal forma que al instanciar de nuevo el servicio 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 memoria 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 la aplicación de nuevo podemos hacer estas acciones para desembocar en la misma situación inicial.

La parte central de este proceso depende de la interfaz configurada en ese momento:

WiFi

```

WifiController: sendbytePacket(), sendbytePacketReliable(), SendBytePacket
                                     (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.

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 comunicación 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 <i>cadena de caracteres</i> 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 <i>double</i> 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 <i>cadena de caracteres</i> para discernir entre mensajes enviados por distintas aplicaciones

Tabla 1.6: 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 1.2) 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 con todos los parámetros es el siguiente:

Campos mensaje desde la aplicación al servicio				
<i>What</i>	<i>arg1</i>	<i>arg2</i>	<i>obj</i>	<i>replyTo</i>
3 = REGISTER_EXCHANGE	0=secundario / 1=primario	0=normal / 1=coordinador	<i>true</i>	Messenger de la aplicación
Extras				
<i>key</i>	<i>value</i>			
appCode	Una <i>cadena de caracteres</i> con el código de aplicación			
nodeName	Una <i>cadena de caracteres</i> con el nuevo nombre del nodo en la red			
periodTask	Un <i>double</i> con el nuevo periodo medido en segundos de la tarea cognitiva a ejecutar			

Tabla 1.7: Actualización de todos los parámetros del servicio

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 sobrescriben 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 añadiríamos otro coordinador, situaciones que no son de interés.

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 y como para minimizar el paso de mensajes por la red no nos pregunta sino que saca una copia de su base de datos los parámetros que tiene el resto de la red son la copia desactualizada de la base de datos del coordinador y no los parámetros nuestros. Esta situación es muy poco frecuente por lo que se ha preferido minimizar el tamaño de mensajes de señalización y el número de estos antes de 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). 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.

Campos mensaje desde un controlador (<i>Bluetooth</i> o <i>WiFi</i>) al servicio				
<i>What</i>	<i>arg1</i>	<i>arg2</i>	<i>obj</i>	<i>replyTo</i>
2 = MESSAGE_READ	Número de <i>bytes</i> leídos	No usado/indiferente	<i>bytes</i> leídos de la red	No usado/indiferente
Extras, controlador: <i>WiFi</i>				
<i>key</i>	<i>value</i>			
device_ip_address	Una <i>cadena de caracteres</i> con la dirección IP del nodo que nos envía los datos			
Extras, controlador: <i>Bluetooth</i>				
<i>key</i>	<i>value</i>			
device_name	Una <i>cadena de caracteres</i> con el nombre proporcionado por los servicios de <i>Bluetooth</i> de <i>Android</i> (no es relevante)			
device_address	Una <i>cadena de caracteres</i> con la dirección MAC de <i>Bluetooth</i> del nodo que nos envía los datos			

Tabla 1.8: Mensaje recibido por la red entregado por ambos controladores al servicio

Una vez tenemos los *bytes* leídos en el servicio los procesamos. Este procesamiento puede ser tan simple como entregar el contenido del mensaje a la aplicación correspondiente o reenviar el mensaje al siguiente eslabón de la cadena (sección explicada en la página 16). 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 1.11) que nos permite recuperar el *messenger* de la aplicación. Así filtramos y entregamos a cada aplicación los mensajes que le interesan. También, gracias al campo “Desde” sabemos quién es el originador del mensaje. Una vez hemos recuperado la aplicación destinataria y el nodo que envió el mensaje, podemos trasladar esta información a la aplicación correspondiente.

Campos mensaje desde el servicio a la aplicación				
<i>What</i>	<i>arg1</i>	<i>arg2</i>	<i>obj</i>	<i>replyTo</i>
8 = INCOMING_CONTENT_PROGRESS	No usado/indiferente	No usado/indiferente	<i>payload</i>	No usado/indiferente
Extras				
<i>key</i>	<i>value</i>			
device_id	Un entero que identifica al nodo que ha enviado el mensaje			

Tabla 1.9: Mensaje datos recibidos

Envío de mensajes

Service: m2service, buildDataMessage(), 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:

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

Tabla 1.10: Mensaje para el envío de datos

Asumimos que el mensaje que se quiere enviar es 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*). Una vez entregado el mensaje al servicio se empieza a procesar. En primer 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 //TODO TODO TODO en el código y manejar el error de casting. o un campo adicional para seleccionar tipo contenido // Si no se superase alguna de estas validaciones se entregaría a la aplicación un mensaje de error con el código y descripción pertinentes y se abortaría el proceso. Si no hay ningún problema, confeccionamos el mensaje que se va a enviar a la red cuyo formato vemos en la tabla 1.11.

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 destinatarios 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 tiempos, el contenido del mensaje y la MAC de *Bluetooth* del nodo que envía el mensaje.

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 1.11: Campos del mensaje de datos

El siguiente punto es ver si podemos enviar el mensaje por la red, para ello debemos estar registrados en una CWSN y no estar en una situación de cambio de contexto, si se da esta situación marcamos el mensaje como “pendiente de envío” en la cola. Si podemos enviarlo lo marcamos como “en envío” y se procede al envío. Para ello necesitamos saber quién o quienes son los siguientes nodos en la cadena de envío, 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 siguiente sección veremos que necesitamos un parámetro más pero que por el momento podemos obviar). Enviar es básicamente 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.

Campos mensaje desde un controlador (<i>Bluetooth</i> o <i>WiFi</i>) al servicio				
<i>What</i>	<i>arg1</i>	<i>arg2</i>	<i>obj</i>	<i>replyTo</i>
3 = MESSAGE_WRITE	número mensajes escritos OK	número mensajes escritos NOK	<i>bytes</i> enviados a través de la red	No usado/indiferente
Extras				
<i>key</i>	<i>value</i>			
requester	Una <i>cadena de caracteres</i> que representa el <i>messenger</i> de la aplicación que origina el mensaje			
addressNotDelivered	Una lista de <i>cadena de caracteres</i> con las MAC de <i>Bluetooth</i> 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 1.12: Mensaje resultado envío de datos desde el controlador al servicio

TODO TODO TODO

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 sí. Si no tuviesemos 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.

Es necesario mensaje ACK En el segundo caso, el mensaje enviado 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 1.12) 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 1.13.

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 1.14) TODO TODO TODO cambiar outgoinf content progress...

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 1.13: Campos del mensaje ACK

Campos mensaje desde el servicio a la aplicación				
<i>What</i>	<i>arg1</i>	<i>arg2</i>	<i>obj</i>	<i>replyTo</i>
9 = OUTGOING_CONTENT _PROGRESS	número men- sajes escritos OK	número men- sajes escritos NOK	<i>payload</i>	No usado/indi- ferente
Extras				
<i>key</i>	<i>value</i>			
IdsNotDelivered	Un array de enteros con los identificadores de los nodos a los cuales no les ha llegado el mensaje			
NamesNotDelivered	Un array de <i>cadena de caracteres</i> con los nombres de los nodos a los cuales no les ha llegado el mensaje			

Tabla 1.14: Mensaje resultado envío de datos presentado a la aplicación

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 obviarnos.

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 envíe 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 sin memoria que 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 1.12 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 1.13) 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

1.5.8. Cambios de contexto

Cambio de contexto con interfaz destino WiFi

Cambio de contexto con interfaz destino Bluetooth

Referencias/Bibliografía