Capítulo 2

Implementación

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

2.1. 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.

2.1.1. Comparación ambas arquitecturas

Clasificar nuestros metodos en los bloques de una arquitectura cognitiva.

2.2. Detalles de implementación

Servicio vs activity.... multi app...
Servicio tipo handlers ->hebra dedicada! hablar de *ANR "Application Not Responding"*
****

El apellido cognitivo supone una abstración del modelo de comunicaciones. La aplicacicion que use de este modelo sólo tiene que entender del dato que quiere enviar o recibir y del destinatario de ese mensaje. En *Android* las aplicaciones son procesos que interactúan con el usuario, están formadas por varios componentes. Entre los distintos que nos provee *Android* el más adeacuado para montar nuestra arquitectura es el servicio. Así nos permite estructurar nuestro modelo de capas dejando todo lo que tenga que ver con el establecimiento, control, transmisión y recepción hubicado en este componente.

(Comentario: Esto creo que es mejor en análisis:
Un servicio es una clase de Java que se ejecuta, en principio, en la misma hebra y en el mismo proceso que la aplicación. Pero que su ciclo de vida está desligado de la pantalla, no tiene interfaz gráfica y está pensado para ejecutarse en *background* con operaciones costosas en el tiempo, que siguen ejecutándose aunque la aplicación que inició estas operaciones no esté activa o esté a otros menesteres. Un servicio es el componente menos dependiente de los posibles eventos que se producen en un móvil: el usuario cambia de aplicación, entra una llamada entrante que hace que el foco cambie...

Además para cumplir el requisito de proveer una serie de operaciones a otras aplicaciones (ya que *Android* nos permite la instanciación de otros componentes fuera de nuestra aplicacion) el único componente que nos permite comunicación en los dos sentidos en cualquier instante de tiempo es el servicio, en concreto entre las configuraciones que nos permite *Android*, elegimos la de *bound service*, es decir un servicio atado, atado a la aplicación que requiere sus servicios, una especie de interfaz cliente-servidor. La interfaz se basa en un objeto *IBdinder* que el cliente (la aplicación) utiliza para comunicarse con el servicio, una referencia que le permite ejecutar los métodos declarados en el servicio. Hay tres formas de declarar esta interfaz:

Extendiendo la clase Binder

Usando un Messenger

Usando AIDL

)

Como se vió en análisis **??**, la forma de recrear la interfaz cliente-servidor necesaria para la confección de un *bound service*, es a través de un *Messenger*. De esta forma la interfaz no declara métodos para su llamada desde el cliente, si no que la comunicación se basa en el intercambio de mensajes que son recogidos por el servicio en su método *handleMessage()* declarado en su *Handler*. La interfaz o el objeto *IBinder* se construye gracias al método *getBinder()* del objeto *Messenger* creado gracias al *Handler* implementado en el servicio.

****

en este caso la aplicación como sólo desea enviar un dato a través de la red cognitiva, si por ejemplo nuestra arquitectura cognitiva estuviese montada sobre una *Activity* (un componente con interfaz gráfica que liga su ciclo de vida a su tiempo en pantalla) en primer lugar la aplicación dejaría de estar en pantalla, con lo cual corre el riesgo de ser destruida y

Dejamos la activity para: las *activities* ligan su ciclo de vida a la interacción en la pantalla, son componentes con interfaz gráfica. Una vez el usuario quita el foco de la aplicación la *activity* es candidata o se produce a su destrucción.

2.3. Implementación de procesos de red

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

2.3.1. Registro de una aplicación en el servicio cognitivo

Código invulcrado

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

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

- `mOut_app`: nos relaciona la clave de `mClients` con el código de aplicación. La aplicación al enviarnos la petición (en la que va incluida su *Messenger*) recuperamos su código de aplicación (gracias al *Messenger*) para poder marcar el mensaje de datos saliente. Sin necesidad de que la aplicación nos informe en cada petición de su código de aplicación.

- `mIn_app`: relaciona el código de aplicación con la clave `mClients`. Al recibir un mensaje leemos el código de aplicación que viene marcado, gracias a este *HashMap* obtenemos la referencia a `mClients` donde obtenemos el *Messenger* para poder entregar el mensaje de datos entrante a la aplicación correspondiente.

2.3.2. Configuración de la interfaz de comunicación

La configuración de cada interfaz de comunicación se realiza en una *AsynTask*: clase diseñada por el sistema *Android* [2]. La peculiaridad de este tipo de tarea es que el núcleo central de ésta se ejecuta en una hebra independiente que es arrancada, manejada y destruida por el propio sistema *Android*, liberando al programador de aplicaciones de esta tarea. Además provee de mecanismos de compartición de información, tanto al principio, como en plena ejecución y también tras la finalización de la tarea entre la *UI-Thread*, desde donde arrancamos la tarea y la hebra donde se ejecuta propiamente ésta, haciendo una transición más fluida. Por esto último ha sido elegida en favor de un *ServiceIntent*.

WiFi: servicio y controlador

Código invulcrado

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

(Comentario: El controlador *WiFi* dependiendo del tipo de nodo en el que esté alojado, tendremos una o dos hebras escuchando para tráfico TCP y UDP. Al configurar la interfaz, liberamos estas hebras, para ello llamamos al método *stop()* del controlador quien llamará a los métodos *cancel()* de cada una de las hebras que cerraran los *sockets* utilizados para la escucha y anulará la suscripción de anuncios del sistema del estado de la conexión entre otros. Una vez liberado todo, las volvemos (hebras y suscripción) a inicializar según proceda, gracias al método *start()*. Esto nos sirve para partir de una situación deseable sin *memory leaks*. //esto yo creo que tb va fuera.)

La esperas señaladas en 1.3.2 consistentes en levantar la interfaz *WiFi* o conectarse a una red *WiFi* se realizan mediante esperas de consulta activa o *polling* cuya variable de control es manejada en el *BroadcastReceiver* de esta interfaz: `es.upm.die.lsi.pfc.CWSNoA.StatusReceiver_wifi`, gracias a una referencia de *callback* que nos permite acceder a las variables deseadas. En el primer caso el anuncio que esperamos tiene la *action* `WIFI_STATE_CHANGED_ACTION` cuyo valor que esperamos es `WIFI_STATE_ENABLED`, la variable de control usada es un booleano llamado `wifiON`. Para la segunda espera esperamos recibir un anuncio del tipo `NETWORK_STATE_CHANGED_ACTION` con el valor `CONNECTED`, la variable usada `waitUntilWifiGetConnected` accedida por métodos *getters* y *setters*. (Comentario: lo de las variables de control... igual lo dejamos fuera no?)

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

Normal Necesita configurar un *ServerSocket* para permitir conexiones entrantes para recibir tráfico del coordinador. Para ello creamos el *socket* de la clase `java.net.ServerSocket` e intentamos vincularlo a un puerto con *bind()* pasandole como parámetro el puerto donde

escuchar. Si este puerto está siendo utilizado para otros propósitos nos devuelve un error del tipo `IOException`. El valor de este puerto es una constante en el código (*Hard-coding*) cuyo valor es '7617' al ser un conocmiento compartido a priori, evitamos tener que transmitir este valor por la CWSN.

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

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

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

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

Bluetooth: servicio y controlador

Código invulcrado

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

(Comentario: Para liberar los recursos, al igual que en el controlador de *WiFi*, llamamos al método *stop()* y restaurando la situación inicial con el método *start()*. //esto podria ser diseño yo creo que va fuera.)
El controlador maneja tres tipos de hebras: la que acepta conexiones (*AcceptThread*), la que lanza la conexión (*ConnectThread*) y la que mantiene la conexión entre el esclavo y el maestro (*ConnectedThread*), de este tipo podemos tener más de una.

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

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

datos y la lista de direcciones MAC para volcar el mensaje a través de la red y llegue al destino oportuno.

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

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

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

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

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

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

2.3.3. Registro de un nodo en la red

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

WiFi

Código invulcrado

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

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

Bluetooth

Código invulcrado

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

La única tipología de red permitida en una red *Bluetooth* como vimos en **??**, es la de estrella, formada por un maestro y varios esclavos (las puntas de la estrella), siguiendo este modelo podemos crear *piconets* [3] identificadas por un UUID necesario para entre otras cosas registrar los *sockets* que van montados sobre un canal *RFCOMM* [4]. Por ello es el nodo coordinador (el maestro) quién lanza las conexiones.

2.3.4. Salida de un nodo de la red

Tras recibir el mensaje de desregistro (ver tabla 1.15), procedemos a borrar la aplicación del mapa `mClients` y con él también las referencias cruzadas en los mapas `mOut_app` y `mIn_app`. Estos procesos se encapsulan en el método *removeClientknownMessengerReply()* que necesita de la representación del objeto *Messenger* en forma de *cadena de caracteres* y *removeClientKnownAppCode()* que requiere del código de aplicación para efectuar el borrado.

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

El envío, en el caso de *WiFi* del paquete *"ByePacket"* se realiza bajo una *AsynTask* finalizando esta con la destrucción del controlador. El envío se realiza mediante un bucle de

conexiones TCP a todos los nodos normales que haya en la red en caso de que el coordinador abandone la red. O sobre el envío bajo UDP (aunque están implementadas ambas formas) al coordinador si es un nodo normal quien abandona la red.

2.3.5. Actualización de parámetros en la red

Código invulcrado

Service: m2service, spreadNodeChangeOnNetwork(), updateInformationToCoordinator()

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

- No se permite actualizar estos parámetros si no es con la aplicación desarrollada en el mismo paquete que el servicio cognitivo, con lo cual el servicio cogería estos valores del archivo de preferencias (usando el objeto *Preference* [5]) ubicado en el mismo paquete que la aplicación y cada vez que se requiera un cambio en el nombre, periodo de la tarea cognitiva o tipo de nodo, tener que abrir la aplicación para cambiar estas.
- La otra opción pasa por permitir que cualquier aplicación pueda cambiar estos valores, garantizando que estas actualizaciones no penalicen a otras aplicaciones que estén en ese momento en ejecución.

Nos hemos decantado por la segunda opción, por ello en diseño vimos que hay una manera especial de actualizar estos parámetros y un mecanismo para garantizar que no haya conflictos. Ello exige comunicar todos los parámetros del servicio a las aplicaciones cuando estas se registran y el almacenaje de estos parámetros en base de datos para independizar el archivo de preferencias que compete exclusivamente a la aplicación desarollada a la vez que el servicio pero que este no debería tener acceso a él.

2.3.6. Intercambio de mensajes

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

Recepción de mensajes

Código invulcrado

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

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

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

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

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

Envío de mensajes

Código invulcrado

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

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

El *LinkedHashMap* tiene como clave un tipo *long* que es el *hash* del mensaje. Para calcular el *hash* no podemos usar la función *hashcode()* de un *byte array* puesto que obtenemos valores distintos en cada invocación [6], por ello necesitamos realizar una función personalizada sencilla pues no preveemos grandes volumenes en esta cola por lo que la colsión es un riesgo bajo. Este *hash* lo calculamos de esta forma[1]:

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

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

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

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

2. Un *byte array* que contiene el mensaje ya codificado listo para su envío.

Utilizando el *hash* del mensaje podemos recuperarlo para cambiar el estado de éste u obtenerlo para su envió a la aplicación tras producirse la confirmación de envío (ver tabla 1.7).

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

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

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

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

Código invulcrado

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

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

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

Cuando detectamos el final de un envío del mensaje en todos los nodos a los que va digirido, cuando devolvemos una respuesta igual que la de *Bluetooth* uniformando de nuevo el

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

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

Forward de mensajes

Código invulcrado

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

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

2.3.7. Sensado del entorno

Parámetro del entorno: RSSI

Código invulcrado

Service: normalCognitiveTask(), mWifiAdapter.getConnectionInfo()

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

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

Parámetro del entorno: Intervalo promedio envío de mensajes

Código invulcrado

Service: normalCognitiveTask()
DataMessageQueue: getAvgArrivalRateUpdateTillNow()

Para calcular el promedio móvil almacenamos los valores a promediar en una *LinkedList*. Los valores a almacenar son de tipo *double*, representa el tiempo transcurrido entre mensajes entragados por las aplicaciones. Este objeto se hubica en la clase `es.upm.die.lsi.pfc`

`.CWSNoA.util.MovingAverage` que es utilizada por la cola de mensajes cada vez que encola un mensaje. En la instanciación se elige el número de muestras de las que se va a tomar el promedio, en nuestro caso el tamaño de ventana es seis.
La razón de poner el promedio en una clase independiente es por si se necesita usar otro tipo de promedios (que puedan ser codificados como *double*) en otra parte de la arquitectura, por ejemplo su uso en políticas cognitivas.

Política cognitiva

Código invulcrado

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

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

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

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

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

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

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

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

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

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

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

2.3.8. Cambios de contexto

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

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

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

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

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.

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

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

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

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

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