Capítulo 1

Diseño

Contar que 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

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				
What	*arg1*	*arg2*	*obj*	*replyTo*
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				
What	*arg1*	*arg2*	*obj*	*replyTo*
3 = REGISTER_EXCHANGE	0=secundario / 1=primario	No usado/indiferente	No usado/indiferente	Messenger de la aplicación
Extras				
key	*value*			
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 = Reciviendo.

Interfaz Un entero cuyo valor 0 representa a *Bluetooth*, el valor 1 representa a *WiFi* y -1 representa a una inerfaz desconicida (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 carácteres* con el nombre del nodo en la red

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

El detalle del mensaje, queda:

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 1.3: 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 delimita a ver si 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:

Cógigo de la aplicación Un *cadena de carácteres* que representa el código de la aplicación

Lista nodos TODO TODO TODO en el código

El detalle del mensaje queda [1]:

[1]Notar 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				
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
nodeNamesList	Array de *cadena de carácteres* 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 *AsynTask* 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 controlladores de las otras interfaces que liberen los recursos (llamando a su método *stop*) para poder partir de una situación deseable.

WiFi: servicio y controlador

SetupWifiInterface (asynTask)
WifiController: stop(), start()

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 controllador 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

SetupBTinterface (asynTask)
BluetoothController: stop(), start(), connect(), setState()

Como en el controllador 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 controllador 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 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 1.3.

1.5.3. Registro de un nodo en la red

acto seguido procedemos al registro del nodo en la red enviando "HelloPackets" (UDP) a la dirección de *broadcast/multicast*

1.5.4. Salida de un nodo de la red

WiFi

voluntaria / no voluntaria

Bluetooth

voluntaria = no voluntaria

1.5.5. Actualización de parámetros en la red

Hablar que en wifi con la entrada se actualizan y en bt no.

1.5.6. Intercambio de mensajes

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

References