UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

ANÁLISIS DEL DESEMPEÑO DE LOS DISPOSITIVOS ANDROID DERIVADO DE DIVERSAS IMPLEMENTACIONES DEL PATRÓN OBJETO ACTIVO.

TESIS

QUE PARA OPTAR POR EL GRADO DE
MAESTRO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:
JOSÉ AURELIO QUEZADA ALVAREZ

Director de Tesis:
Dr. Jorge Luis Ortega Arjona
Departamento de Matemáticas, Facultad de Ciencias.

Ciudad Universitaria, Cd. Mx. octubre 2016

Índice general

Agradecimientos

Quiero agradecer a aquellos que hicieron esto posible, primeramente, al doctor Ortega, por su tiempo y paciencia, pero sobre todo por mostrar interés real en mi tema de investigación, y por sus muy interesantes e inspiradoras clases, las cuales siempre disfruté y recordaré.

Segundo, quiero agradecer a mi madre, quien me ha ayudado en distintos sentidos, y a pesar de nuestras diferencias, siempre ha estado ahí.

En tercer lugar, y no por ello menos importante, agradezco a todos mis profesores, y a mis compañeros: Heriberto, Octavio, Ernesto, Alex, Mario, Víctor, Fabiola y Karen, solo ustedes saben por lo que pasamos.

Finalmente, quiero agradecer al CONACYT, por el apoyo económico sin el cual simplemente no habría podido dedicarme al posgrado.

Capítulo 1

Introducción

1.1. El contexto

Con el creciente desarrollo de dispositivos móviles como laptops, smartphones, etcétera, ha surgido un área conocida como "Cómputo Móvil". Los dispositivos involucrados en esta área, al igual que los dispositivos tradicionales, utilizan software para su funcionamiento. Sin embargo, requieren de ciertos elementos de software orientados a satisfacer algunas características específicas, (como el procesamiento concurrente) dentro de un ambiente de movilidad física.

Por otro lado, a través de la experiencia, se han identificado una serie de modelos que resultan satisfactorios en la resolución de problemas recurrentes bien definidos en la construcción de software. Los llamados "Patrones de Diseño" y "Patrones Arquitectónicos de Software" son descripciones de elementos estructurales, adaptados a resolver un problema de diseño general en un contexto particular [1].

Una de las principales características de los dispositivos móviles en las que se centra la atención de los Patrones de Diseño es la capacidad de procesamiento concurrente. La razón es que, a través de ella, se puede mejorar el desempeño de un dispositivo, y así, mejorar la experiencia de usuario.

Incluso antes de la proliferación de los dispositivos móviles, se identificaron Patrones de Diseño que manejan tareas concurrentes, los cuales a su vez están compuestos de patrones más especializados. Tal es el caso del patrón "Objeto Activo", enfocado en desacoplar el método de ejecución del método de invocación en programación Orientada a Objetos. Esto para mejorar la

concurrencia y simplificar el acceso sincronizado a objetos que residen en su propio hilo de control. El patrón Objeto Activo está compuesto de algunos otros patrones y elementos que pueden ser implementados de diversas maneras, siempre que cumplan con su funcionalidad. De ahí que el patrón en los sistemas operativos Android posea una implementación particular que no es necesariamente la mejor para obtener el desempeño deseado.

1.2. El problema

El control de accesos concurrentes a objetos compartidos en Android utiliza el patrón de diseño Objeto Activo. Sin embargo, la implementación de sus diversos elementos es muy simple, por lo que vale la pena preguntarnos: ¿Hay alguna implementación distinta del patrón Objeto Activo, que al incluirla en Android incremente el desempeño (en términos de tiempo de respuesta) de los dispositivos?

Como la implementación de un Patrón de Diseño puede realizarse de distintas maneras, siempre que cumpla con la estructura, participantes, colaboraciones y consecuencias que ha sido definido, se cuenta con un grado de libertad que da la posibilidad de realizar implementaciones adecuadas. Pero al mismo tiempo, se puede caer en implementaciones que entorpecen la ejecución del software.

Al revisar la implementación del patrón Objeto Activo utilizado en el sistema operativo Android, se puede suponer que existe una implementación distinta que sea más eficiente. El "Proxy", por ejemplo, está implementado de una manera muy simple. Visto individualmente, funciona como un simple mecanismo para encolar las peticiones o "Method Requests", y no toma desiciones o realiza tratamiento alguno con ellas. Existen algunas variantes del patrón como por ejemplo el "Synchronization Proxy" ó el "Cache Proxy". Esto puede implicar el paso de Method Requests a los Objetos Activos de manera deficiente. La "Activation List" (de igual forma) está implementada como una cola de la que se extraen las Method Requests en el orden en que fueron encoladas. Sin embargo, se podría implementar de manera más sofisticada para extraerlas de una manera más provechosa. El "Scheduler" funciona de manera muy simple: ejecuta un ciclo infinito para desencolar las Method Requests en el orden en que se encolaron.

1.3. La hipótesis

Al adecuar la implementación original del patrón Objeto Activo en Android para obtener una de sus variantes denominada Objeto Activo Multihilos (o Thread Pool Active Object en inglés) es posible incrementar su desempeño. Al contar con múltiples hilos, se pueden ejecutar varias tareas concurrentemente, eliminando la necesidad de tener que esperar la finalización de una para comenzar con la siguiente, además de limitar la cantidad de hilos que se pueden crear simultáneamente, evitando la sobrecarga de trabajo del procesador.

1.4. La aproximación

En busca de obtener una implementación del patrón Objeto Activo Multihilos, se debe tener en cuenta cuál es el Scheduler, ya que dichos hilos se suelen administrar desde él por medio de un arreglo de estos objetos, conocido como Thread Pool. En Android el Scheduler del patrón es la clase Looper de las APIs de JAVA, por lo que se le debe agregar un atributo que almacene ese arreglo y modificar el o los métodos que desencolan las peticiones para que las envíe a ejecución en estos hilos.

1.5. Contribuciones

Para lograr el objetivo de esta tesis, que es obtener una implementación del patrón Objeto Activo que maneje las actividades concurrentes de tal manera que aumente el desempeño del sistema operativo Android, se contribuye con los siguientes puntos:

- Un análisis de las deficiencias del tratamiento de concurrencia en la implementación original del sistema operativo Android.
- Una versión modificada del código que implementa el patrón Objeto Activo en Android 5.1.1 Lollipop, que mejore su tiempo de ejecución y desempeño con respecto a la implementación original, incrementando así su responsividad y mejorando la experiencia de usuario.

1.6. Estructura de la tesis

El presente trabajo está estructurado en 6 capítulos, los cuales se describen a continuación.

1. **Antecedentes.**

   En este capítulo se presentan las bases necesarias para comprender el desarrollo realizado en busca de la comprobación de la hipótesis y la obtención de las contribuciones. En concreto, se describen los conceptos necesarios del área de Patrones de Software y se presenta el patrón Objeto Activo. A continuación, se describe la arquitectura de Android para conocer específicamente en donde se realizan los cambios propuestos, y se presenta de manera relativamente profunda la descripción del código que forma el patrón Objeto Activo en Android 5.1.1, mencionando algunas deficiencias. Y finalmente se introduce el término de Métrica de Software, para definir un par de métricas necesarias para los fines del presente trabajo.

2. **Trabajo relacionado.**

   Aquí se describe el trabajo realizado en torno al análisis y uso del patrón Objeto Activo. Primeramente, se presenta el trabajo titulado "Resource Management for Mobile Operating Systems based on the Active Object Model". En este trabajo se plantea la necesidad de algún modelo para la administración de recursos eficiente y oportunamente que permita conocer por lo menos, las restricciones de los dominios de aplicación emergentes que son altos consumidores de recursos en los dispositivos móviles, y como respuesta se presenta un esquema para integrar las técnicas de administración de recursos con el modelo de concurrencia de los sistemas operativos embebidos, que usan el modelo de concurrencia de Objeto Activo, utilizando como ejemplo el sistema operativo Symbian.

   Y como segundo trabajo se presenta "Pattern-Based Simulation: Simulating the Actor Model Using the Active Object Behavioural Pattern". Se propone el uso del patrón Objeto Activo en combinación con técnicas de simulación estocástica, para producir un modelo de simulación que

asista a los diseñadores de software en la estimación y el pronóstico de las propiedades de desempeño de un programa concurrente. El modelo de simulación presentado recibe como entrada el patrón de comportamiento de un Objeto Activo y elementos de teoría de colas, y produce como salida estimaciones acerca del comportamiento de desempeño del Objeto Activo.

3. **Análisis del desempeño de los dispositivos Android derivado de diversas implementaciones del patrón Objeto Activo.**

   En este capítulo se presentan las deficiencias de implementación del patrón Objeto Activo utilizado para control de concurrencia sobre objetos compartidos en el sistema operativo Android. Además se presenta el código de la implementación propuesta.

4. **Experimentación**

   En este capítulo, se describe el experimento a que se deben someter las distintas implementaciones analizadas para poder dilucidar si la propuesta corrobora la hipótesis o demuestra lo contrario. Se describen tanto las actividades del proceso como los elementos involucrados en él, los datos de entrada, factores controlables e incontrolables, y los datos de salida. Además, se describe la forma en que es posible obtener los datos desde el código programado y la herramienta desarrollada para llevar a cabo el experimento: una aplicación de Android en la cual se cargan los datos de entrada, se especifican y envían las peticiones al Objeto Activo, y finalmente muestra los resultados. Por último, se describen las características del ambiente donde se realizan los experimentos.

5. **Medición del desempeño del patrón Objeto Activo en las distintas implementaciones analizadas.**

   Al realizar el experimento, se recopilan medidas de los tiempos de ejecución de las distintas implementaciones y, a partir de la medición se obtienen algunas métricas. En este capítulo se presenta dichas métricas que resultan clave para concluir si la hipótesis es falsa o cierta, y si la alternativa propuesta puede ser considerada como la contribución planteada de brindar una versión modificada del código que implementa el

patrón Objeto Activo en Android 5.1.1 Lollipop, que mejore el desempeño con respecto a la implementación original. Las métricas obtenidas de la versión original y las modificadas se presentan en forma tabular y se contrastan gráficamente.

6. Conclusiones

Una vez que se obtuvieron las métricas de desempeño de la implementación original y la propuesta, se presentan las conclusiones obtenidas. Al final se mencionan propuestas de trabajo futuro.

Capítulo 2

Antecedentes

Para establecer el contexto en donde se aplicarán las propuestas de solución al problema identificado, en el presente capítulo, se abordan primeramente los "Patrones de Software", área de la Ingeniería del Software que describe soluciones generales para problemas identificados que se presentan constantemente, y para los cuales se han proveído soluciones que han demostrado ser eficaces.

Debido a que cada autor tiene una visión de los puntos que deben ser cubiertos al describir un patrón, existen varias formas de hacerlo. Por lo tanto aquí se incluye la descripción de la forma POSA (Pattern Oriented Software Arquitecture [3]), la cual es una fuente bibliográfica muy popular en el área.

Dentro de los Patrones de Software, existe uno utilizado para el manejo de tareas concurrentes: el patrón Objeto Activo, que desacopla la invocación de los métodos de su ejecución mediante el almacenamiento temporal de los mensajes enviados a los objetos que poseen tales métodos. Esos objetos se ejecutan en hilos distintos al del código cliente. Este patrón se utiliza en Android, el cual es un sistema operativo actualmente utilizado en diversos dispositivos móviles y es de código abierto, por lo tanto es susceptible de análisis y mejora.

2.1. Patrones de Software

Según menciona Christopher Alexander, "Cada patrón describe un problema que ocurre una y otra vez en un entorno, y a continuación describe el núcleo de la solución a tal problema, de tal manera que se puede usar esta

solución un millón de veces, sin tener que hacerlo de la misma manera dos veces" [8].

Dentro de la literatura de patrones de software, se acostumbra relacionar un problema y su solución mediante un contexto, es decir, que si se tiene un problema bajo ciertas circunstancias, entonces se puede aplicar tal solución. Por ejemplo el mismo Alexander incluye el contexto en su definición de patrón [9]:

> Cada patrón es una regla de tres partes, la cual expresa la relación entre un cierto contexto, un problema y una solución.

Se incluye el contexto dado que pueden existir muchas soluciones al problema planteado, pero bajo ciertas circunstancias las opciones se acotan y resulta evidente la necesidad de utilizar "una" solución. Al contar con un problema bajo las circunstancias identificadas en las que aplica esa solución es cuando se manifiesta el patrón.

Se pueden distinguir tres grupos: patrones arquitectónicos, patrones de diseño y modismos.

- "Un patrón arquitectónico expresa un esquema de organización estructural fundamental para sistemas de software. Proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades, e incluye reglas y directrices para la organización de las relaciones entre ellos. Los patrones arquitectónicos son plantillas para arquitecturas de software concretas. Especifican las propiedades estructurales de todo el sistema de una aplicación, y tienen un impacto en la arquitectura de sus subsistemas. La selección de un patrón arquitectónico es, por lo tanto, una decisión de diseño fundamental cuando se desarrolla un sistema de software" [3].

- "Un patrón de diseño proporciona un esquema para refinar los subsistemas o componentes de un sistema de software, o las relaciones entre ellos. Describe una estructura de comunicación entre componentes comúnmente-recurrente que resuelven un problema de diseño general en un contexto específico" [1].

"Los patrones de diseño son patrones de escala mediana. Son patrones más pequeños en escala que los patrones arquitectónicos, pero tienden a ser independientes de un lenguaje de programación particular o paradigma de programación. La aplicación de un patrón de diseño no tiene efecto en la estructura fundamental de un sistema de software, pero puede tener una fuerte influencia en la arquitectura de un subsistema" [3].

- "Un modismo es un patrón de bajo nivel específico de un lenguaje de programación. Un modismo describe cómo implementar los aspectos particulares de los componentes o las relaciones entre ellos utilizando las características de un lenguaje dado. Los modismos representan los patrones de más bajo nivel. Estos abordan aspectos tanto de diseño como de implementación [3]."

2.1.1. La forma POSA

Existen varias formas de describir los Patrones de Software. En esencia, todas incluyen las mismas secciones, pero varían un poco dependiendo de él o los autores.

En las siguientes secciones se utiliza la forma POSA para describir los patrones clave de esta tesis, incluyendo las secciones pertinentes para los fines de esta tesis. La forma POSA contempla las siguientes secciones [3]:

Nombre: El nombre y un resumen corto del patrón.

También conocido como: Otros nombres del patrón, si hay algún otro conocido.

Ejemplo: Un ejemplo del mundo real en donde se demuestre la existencia del problema y la necesidad del patrón.

Contexto: La situación en la cual el patrón se aplica.

Problema: El problema que el patrón resuelve, incluyendo una discusión de sus fuerzas asociadas.

Solución: El principio fundamental de la solución que sirve como base al patrón.

Estructura: Una especificación detallada de los aspectos estructurales del patrón, que incluye un diagrama de clases en OMT [12].

Dinámica: Escenarios típicos que describen el comportamiento en tiempo de ejecución del patrón. Los escenarios se ilustran con diagramas de secuencia de mensajes entre objetos.

Implementación: Guías para la implementación del patrón.

Ejemplo resuelto: Discusión sobre cualquier aspecto importante para resolver el ejemplo que no se haya cubierto en las secciones Solución, Estructura, Dinámica e Implementación.

Variantes: Una breve descripción de las variantes o especializaciones de patrón.

Usos conocidos: Ejemplos de usos del patrón, tomados de sistemas existentes.

Consecuencias: Los beneficios que provee el patrón, y cualquier potencial desventaja.

Véase también: Referencias a patrones que resuelven problemas similares, y a patrones que ayudan a refinar el patrón que se describe.

2.1.2. El patrón Objeto Activo

El patrón de diseño Objeto Activo desacopla el método de ejecución del método de invocación para mejorar la concurrencia y simplificar los accesos sincronizados a objetos que residen en su propio hilo de control [4].

Tambien conocido como

Objeto concurrente [4].

Contexto

Existen clientes que acceden a objetos que corren en hilos de control separados [4].

Problema

Muchas aplicaciones se benefician del uso de objetos concurrentes para mejorar su calidad de servicio, por ejemplo, permitiéndole a una aplicación controlar múltiples solicitudes de clientes simultáneamente. Como esos objetos son compartidos y modificados, el acceso debe ser controlado y sincronizado [4]. Las fuerzas que debemos balancear al implementar el patrón son:

- Los métodos de procesamiento intensivo invocados en un objeto concurrentemente no deben bloquear el proceso entero, degradando así la calidad del servicio de otros objetos concurrentes.
- El acceso sincronizado a los objetos compartidos debe ser sencillo de programar. En particular, las invocaciones de métodos clientes de un objeto compartido que está sujeto a restricciones de sincronización deben ser serializadas y programadas (agendadas en tiempo) transparentemente.
- Las aplicaciones deben ser diseñadas para utilizar el paralelismo disponible en una plataforma de hardware/software transparentemente.

Solución

La solución pasa por desacoplar el método de invocación, del método de ejecución para cada uno de los objetos expuestos a las fuerzas descritas. El método de invocación ocurre en el hilo de control del cliente, mientras que el método de ejecución debe ocurrir en un hilo separado. El diseño del desacople debe ser de tal modo que parezca que el cliente invoca un método ordinario.

Para realizar lo descrito, se utiliza un objeto “Proxy”, que es la interface (medio de comunicación) de un Objeto Activo y un ”Servant”provee su implementación. Ambos objetos se ejecutan en hilos separados (el Proxy se ejecuta en el hilo del cliente), por lo que las invocaciones y ejecuciones se

pueden realizar concurrentemente.

En la ejecución, el Proxy transforma las invocaciones de métodos de los clientes en "Method Requests"(peticiones de ejecución de métodos) y las almacena en una "Activation List", es decir, una lista o cola. Por medio de un "Scheduler", se provee de algún método de ordenación o priorización para la ejecución de las Method Requests. Un evento cíclico del Scheduler se ejecuta continuamente en el hilo tomando las Method Requests de la Activation List y despachándolas. Los clientes obtienen los resultados de la ejecución por medio de un objeto denominado "Future" (Futuro), retornado por el Proxy [4].

Estructura

El diagrama de clases del patrón Objeto Activo se muestra en la figura 2.1. El patrón consiste de seis componentes [4]:

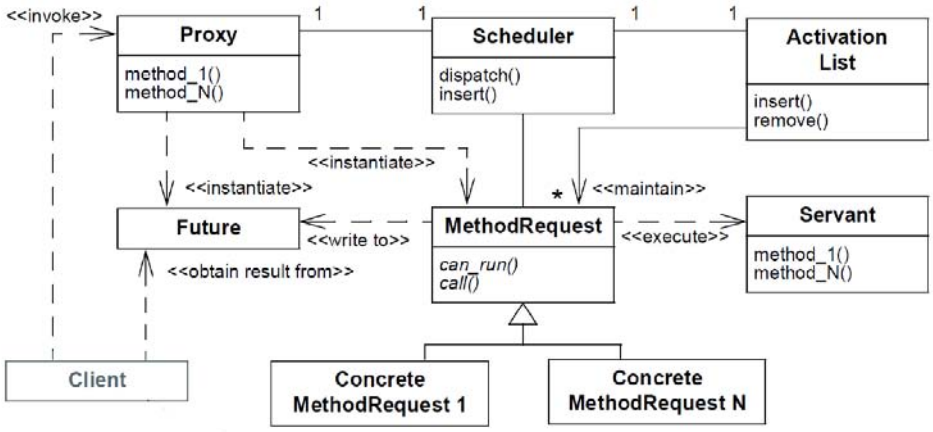

Figura 2.1: Diagrama de clases del patrón Objeto Activo [4].

Proxy. Provee una interface que permite a los clientes invocar los métodos públicos de un Objeto Activo. Este reside en el hilo del cliente [4].

Method Request. Es creado cuando un cliente invoca un método definido por el Proxy. Consiste de la información del contexto, como los parámetros de un método necesarios para la ejecución de una invocación de método específico, y regresar el resultado al cliente. La clase de un Method Request define una interface para ejecutar los métodos de un Objeto Activo. Esta interface también contiene "Guard Methods"(Métodos Guardias) que pueden

ser utilizados para determinar cuándo un Method Request puede ser ejecutado. Para cada método público ofrecido por un Proxy que requiere acceso sincronizado en el Objeto Activo. La clase de Method Request es sub clasificada para crear una clase de Method Request concreta [4].

Activation List. Este objeto mantiene un búfer limitado de Method Requests pendientes, creadas e insertadas por el Proxy. Además, mantiene el registro de las que pueden ser ejecutadas. La lista desacopla el hilo cliente donde reside el cliente del hilo donde se ejecuta el método Servant, así que los dos hilos se pueden ejecutar concurrentemente [4].

Scheduler. Este objeto decide cuál Method Request ejecutar a continuación. Esta decisión se basa en varios criterios, como el orden de inserción o el estado del Objeto Activo, por ejemplo [4].

Servant. Define el comportamiento y el estado que se modela como un Objeto Activo. Los métodos que implementa corresponden con la interface del Proxy y con las Method Requests que el Proxy crea. También puede contener otros métodos que las Method Requests pueden utilizar para implementar sus Guards. Un método del Servant es invocado cuando su Method Request asociada es ejecutada por un Scheduler, y por lo tanto se ejecuta en el hilo de ese Scheduler [4].

Future. Su función es permitir al cliente obtener el resultado de la invocación de un método después de que el Servant finalice la ejecución del mismo. Este objeto es creado cuando un cliente invoca un método en un Proxy. Cada Future almacena espacio por cada método invocado para almacenar su resultado. Cuando un cliente desea obtener el resultado, puede aplicar rendzevous con el Futuro mediante bloqueo o sondeo hasta que el resultado sea obtenido y almacenado en el Future [4].

Dinámica

El comportamiento del patrón se divide en tres partes, que se muestran en la figura 2.2 [4]:

1. **Construcción del Method Request y planeación**. Cuando un cliente invoca un método en el Proxy, éste crea un Method Request y lo pasa a su Scheduler, que lo encola en la Activation List. Si el método retorna algún valor, un objeto Futuro es retornado al cliente [4].

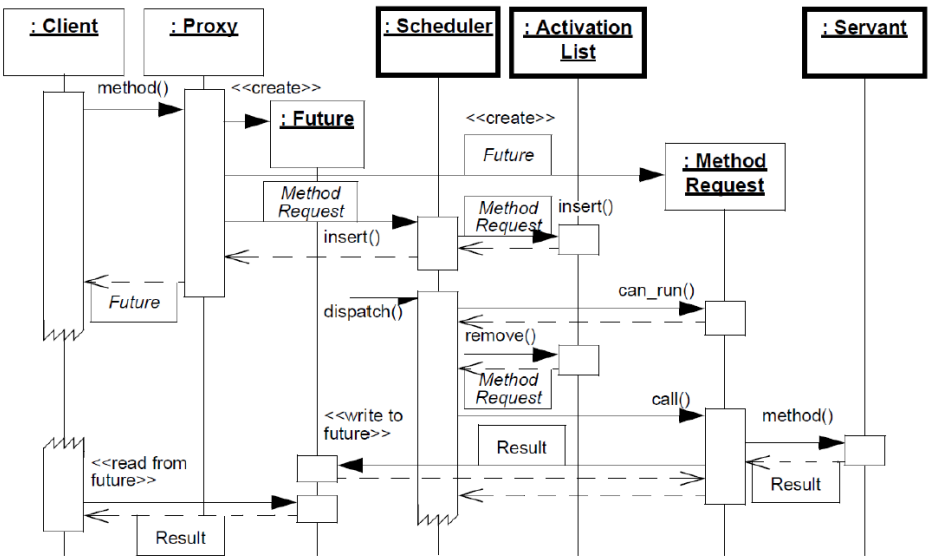

Figura 2.2: Diagrama de objetos del patrón Objeto Activo [4].

2. **Ejecución de Method Request**. El Scheduler de un Objeto Activo se ejecuta contínuamente en un hilo diferente al de sus clientes, y revisa la Activation List para determinar cuáles Method Requests pueden ser ejecutadas invocando el correspondiente método Guard. Cuando un método puede ser ejecutado, el Scheduler lo remueve de la Activation List y ejecuta el método vinculado en el Servant, creando el valor de retorno, si es el caso [4].

3. **Finalización**. En el último paso, el resultado, si es que hay, es almacenado en el Futuro de tal forma que al aplicar rendezvous, el cliente puede obtenerlo. El Scheduler del Servant vuelve a monitorear la Activation List para buscar más Method Requests ejecutables. Los objetos Method Request y Future pueden ser eliminados [4].

Variantes

Objeto Activo Multi-hilos

Es una generalización del Patrón Objeto Activo que soporta múltiples hilos Servant por cada Objeto Activo para incrementar el desempeño y la responsividad. Mientras no se encuentre procesando Method Requests, cada hilo Servant del grupo se bloquea en una Activation List. Tan pronto como

un Method Request esté listo para ejecución, el Scheduler del Objeto Activo la asigna a un hilo Servant [4].

Un grupo de hilos Servant comparte una sola implementación de Servant, por lo que esta variante debe proteger el estado interno de los métodos Servant mediante algún mecanismo de sincronización como un mutex [4].

Consecuencias

En cuanto a los beneficios del patrón, se tienen los siguientes:

- Al aplicar el patrón Objeto Activo, se mejora la concurrencia al permitir a los hilos clientes y la ejecución de métodos asíncronos correr simultáneamente. La complejidad de sincronización se simplifica usando un Scheduler que evalúa las restricciones de sincronización para garantizar el acceso serializado a los servidores de acuerdo a su estado [4].
- Se aprovecha transparentemente el paralelismo disponible si el software y el hardware soportan múltiples procesadores eficientemente, ya que el patrón le permite a múltiples objetos activos correr en paralelo, solo limitados por sus restricciones de sincronización [4].
- El orden de ejecución puede diferir del orden de invocación, ya que los métodos invocados asíncronamente se ejecutan de acuerdo a sus restricciones de sincronización definidas en Guards y a las políticas de planeación. Este desacople puede mejorar la concurrencia y flexibilidad de una aplicación [4].

Las posibles desventajas relacionadas con el patrón son:

- Dependiendo de la implementación de los objetos activos, puede ocurrir un sobreuso de recursos por el intercambio del contexto, la sincronización y el traslado de datos, cuando se planean y ejecutan las invocaciones de métodos. El consejo es no utilizar objetos activos muy refinados [4].
- Puede resultar difícil de depurar el código que utiliza el patrón debido a la concurrencia y no determinismo de los varios Schedulers de Objetos Activos y al hilo planificador del sistema operativo subyacente. Aunque los Guards determinan el orden de ejecución, éstos pueden ser difíciles de entender y depurar. Incluso, si no están adecuadamente implementados, puede ocurrir una situación en la que algunas Method Requests nunca se ejecuten [4].

2.2. El sistema operativo Android

El sistema operativo Android es uno de los principales sistemas operativos para dispositivos móviles, utilizado en teléfonos inteligentes, tabletas y recientemente en los llamados wearables, como los relojes inteligentes. Android es una plataforma de desarrollo libre, basada en el sistema operativo Linux y de código abierto, es decir que se puede usar y modificar sin pagar regalías por derechos de autor. Además, asegura la portabilidad de las aplicaciones a una gran variedad de dispositivos, ya que generalmente se desarrollan en JAVA, por lo que se hace uso de la máquina virtual creada para este lenguaje. Otra opción es crear aplicaciones en C/C++ [6]. En la actualidad es posible desarrollar aplicaciones en C# gracias a la máquina virtual de MONO.

2.2.1. La arquitectura de Android

Android está estructurado, como se observa en la figura 2.3, por cuatro capas. Una característica importante es que todas las capas están basadas en código libre [6].

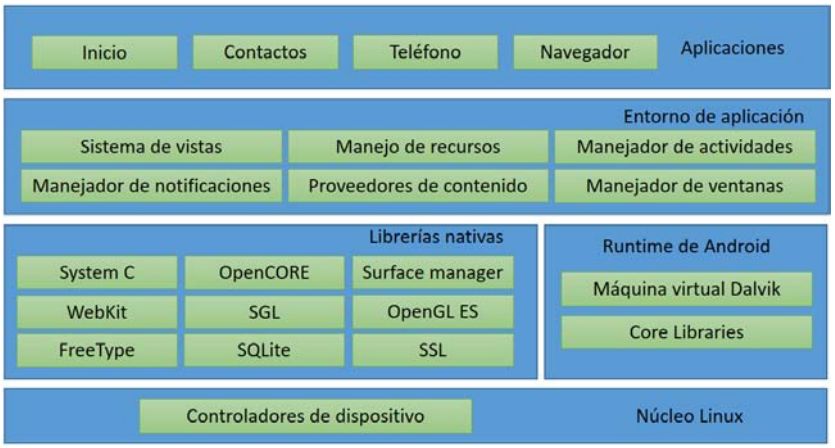

Figura 2.3: Arquitectura de Android

Núcleo

El núcleo de Android es el sistema operativo LINUX versión 2.6. Este proporciona servicios como seguridad, manejo de la memoria, multiproceso,

pila de protocolos y soporte para drivers de dispositivos. Ésta capa actúa como capa de abstracción entre el hardware y el resto de la pila. Por lo tanto, es la única dependiente del hardware [6].

Runtime de Android y Librerías nativas

El Runtime de Android está basado en la máquina virtual de JAVA. Sin embargo, debido a las limitaciones de los dispositivos móviles, Google ha desarrollado una nueva versión denominada Dalvik, para un mejor funcionamiento. Para esto, ejecuta sus propios archivos Dalvik, los cuales están optimizados para ahorrar memoria. Además, está basada en registros. Cada aplicación corre en su propio proceso Linux con su propia instancia de Dalvik, delegando al kernel de Linux algunas funciones, como el manejo de hilos y el manejo de la memoria a bajo nivel. También se incluye en el Runtime de Android "Core libraries", que es un conjunto de la mayoría de las librerías de JAVA.

En la misma capa del Runtime, se encuentran las Librerías nativas. Son un conjunto de librerías en C y C++ utilizadas en algunos componentes de Android. Están compiladas en código nativo del procesador, y muchas utilizan proyectos de código abierto. Algunas de ellas sirven para crear gráficos en 2D y 3D. Algunas son utilizadas como motor de bases de datos relacionales, etcétera [6].

Entorno de aplicación

Es una plataforma de desarrollo para aplicaciones. Mediante ella se pueden utilizar los sensores del equipo, el servicio de localización, la barra de notificaciones, etcétera. Esta capa está diseñada para simplificar la reutilización de componentes, ya que en ella las aplicaciones pueden publicar sus capacidades, y otras usarlas si cumplen las restricciones de seguridad.

El entorno de aplicación utiliza el lenguaje de programación JAVA y los servicios más importantes que incluye son [6]:

- **Views**. Es un conjunto de vistas que son la parte visual de los componentes.
- **Resource Manager**. Proporciona acceso a recursos que no son código.
- **Activity Manager**. Maneja el ciclo de vida de las aplicaciones y proporciona un sistema de navegación entre ellas.

- **Notification Manager**. Permite a las aplicaciones mostrar alertas personalizadas en la barra de estado.
- **Content Providers**. Es un mecanismo sencillo para acceder a datos de otras aplicaciones, por ejemplo los contactos.

Aplicaciones

En esta capa se ubican todas las aplicaciones instaladas en un sistema Android: aplicaciones de propósito general como mensajeros, visores de imágenes, reproductores de audio, etcétera. Todas se deben ejecutar en la máquina virtual Dalvik, para garantizar la seguridad del sistema [6].

2.3. Descripción del patrón Objeto Activo en Android, sus componentes y sus deficiencias.

Después de describir el patrón Objeto Activo en general, sus componentes y el comportamiento de los mismos, y el sistema operativo Android, ahora, se aborda la implementación del patrón en Android 5.1.1 Lollipop, objeto principal de estudio de esta tesis. Los componentes están codificados mediante las siguientes clases en las API's del lenguaje JAVA en la capa de Entorno de aplicación:

- **Handler.** Es el componente que funge como Proxy, ya que a través de él se obtienen objetos Message, y después los recibe como Method Requests de objetos clientes.
- **Message.** Almacena el mensaje que envía una aplicación cliente, que es un Method Request.
- **MessageQueue.** En esta clase, se almacenan los Method Request de objetos clientes, por lo tanto es considerada la Activation List.
- **Looper.** Es el Scheduler, pues envía a ejecución los Method Request pendientes.

A continuación, se describe a más detalle cada uno de estos elementos.

Handler

En una aplicación Android, en la que se desea desacoplar la invocación de un método de la ejecución del mismo, por ejemplo, para actualizar la interfaz de usuario mientras se espera el resultado. Las Method Requests se envían a una instancia de la clase Handler, ya que ésta se crea en el hilo de ejecución principal de la aplicación, pero permite crear hilos adicionales que ejecuten los métodos solicitados. Así, Android ejecuta varias tareas simultáneamente.

La clase Handler permite enviar y procesar objetos Message y Runnable asociados con la MessageQueue de un hilo. Cada instancia de Handler está asociada con un solo hilo y esa cola de mensajes del hilo. Cuando se crea un nuevo Handler, se limíta al hilo/cola de mensajes del hilo que la está creando. A partir de ese momento, entrega mensajes y ejecutables a la cola de mensajes, y los ejecuta conforme vayan saliendo de la cola.

Hay dos usos principales para un Handler: (1) programar mensajes y ejecutables a ser ejecutados en un punto en el futuro; y (2) encolar una acción a ser ejecutada en un hilo diferente al propio.

La ejecución de mensajes se logra mediante los métodos post(Runnable), postAtTime(Runnable, long), postDelayed(Runnable, long), sendEmptyMessage(int), sendMessage(Message), sendMessageAtTime(Message, long), y sendMessageDelayed(Message, long). Las versiones de post permiten encolar objetos Runnable a ser llamados por la cola de mensajes cuando son recibidos; las versiones sendMessage permiten encolar un objeto Message que contiene un conjunto de datos que se procesan por el método handleMessage(Message) de Handler (que requiere que se implemente una subclase de Handler).

Al publicar o enviar a un Handler, se puede o bien permitir al elemento ser procesado tan pronto como la cola de mensajes esté lista para hacerlo, o especificar un retraso antes de que sea procesado, o un momento absoluto para ser procesado. Las últimas dos le permiten implementar timeouts, ticks, y otro comportamiento basado en tiempo.

Cuando un proceso se crea, su hilo principal es dedicado a correr una cola de mensajes que se dedica a administrar los objetos de aplicación de nivel superior y cualquier ventana que creen. Es posible crear otros hilos, y comunicarlos con el hilo de la aplicación principal a través de un Handler. Esto se hace llamando los mismos métodos post o sendMessage como antes, pero desde su nuevo hilo. El Runnable o Message dado es entonces coloca-

do en la cola de mensajes de Handler, y procesado cuando sea apropiado [15].

Message

Define un mensaje que contiene una descripción y datos arbitrarios que pueden ser enviados a un Handler. Este objeto contiene dos campos: int extras y un campo objeto extra, que le permite no hacer asignaciones en muchos casos.

Mientras que el constructor de Message es público, la mejor manera de obtener uno de éstos es llamar a Message.obtain(), o uno de los métodos Handler.ObtainMessage(), que los obtiene de un conjunto de objetos reciclados [15].

MessageQueue

Es una clase de bajo nivel que mantiene la lista de mensajes a ser despachados por el Looper. Los mensajes no son agregados directamente a MessageQueue, sino a través de objetos Handler asociados con el propio Looper. Se puede obtener la MessageQueue para el hilo actual con el método Looper.myQueue() [15].

Looper

Esta clase se utiliza para ejecutar un ciclo de mensajes para un hilo. Los hilos por defecto no tienen un ciclo de mensajes asociado a ellos; para crear uno, se llama a prepare() en el hilo del ciclo, y después se invoca el método loop() para procesar los mensajes, hasta que se detenga el hilo.

La mayor parte de la interacción con el ciclo de mensajes es a través de la clase Handler. Un ejemplo típico de la implementación de un hilo Looper se presenta en la figura 2.4, usando la separación de prepare() y loop() para crear un Handler inicial para comunicarse con el Looper [15].

Al igual que en la sección 2.1.2, en la figura 2.5 se presenta el diagrama de clases del patrón Objeto Activo, ahora con los detalles específicos de la implementación en Android 5.1.1.

Y la dinámica de ésta implementación se muestra en los diagramas de secuencia de las figuras 2.6 y 2.7. El primero de ellos muestra cómo se crean y encolan los mensajes (los métodos principales de los mensajes que se envían

```
class LooperThread extends Thread {
    public Handler mHandler;

    public void run() {
        Looper.prepare();

        mHandler = new Handler() {
            public void handleMessage(Message msg) {
                // process incoming messages here
            }
        };

        Looper.loop();
    }
}
```

Figura 2.4: Ejemplo de implementación de hilo Looper [15].

entre objetos se muestran en rojo). El segundo diagrama muestra cómo se desencolan y ejecutan los mensajes que ya se encuentran almacenados en la Activation List (clase MessageQueue). Los métodos mostrados son los necesarios para realizar el flujo básico. Sin embargo, cada una de las clases contiene más métodos, algunos de los cuales son variantes de los mostrados, y sirven por ejemplo, para colocar mensajes al frente de la Activation List o para agregarles un retraso de ejecución. La lista completa puede ser consultada en [15].

Ahora bien, los componentes que se muestran en los diagramas cumplen su función. Sin embargo, al analizar su implementación en Android 5.1.1 Lollipop con el lenguaje JAVA, se observan ciertos aspectos susceptibles de modificar para un mejor aprovechamiento de los recursos. Para evidenciar esto, a continuación se describe el proceso general mediante el código relevante al utilizar el patrón en Android.

De acuerdo a los diagramas de secuencia de las figuras 2.6 y 2.7, primero se crea un cliente, por ejemplo, la clase MainActivity de una aplicación cualquiera:

```
public class MainActivity extends AppCompatActivity{//...}
```

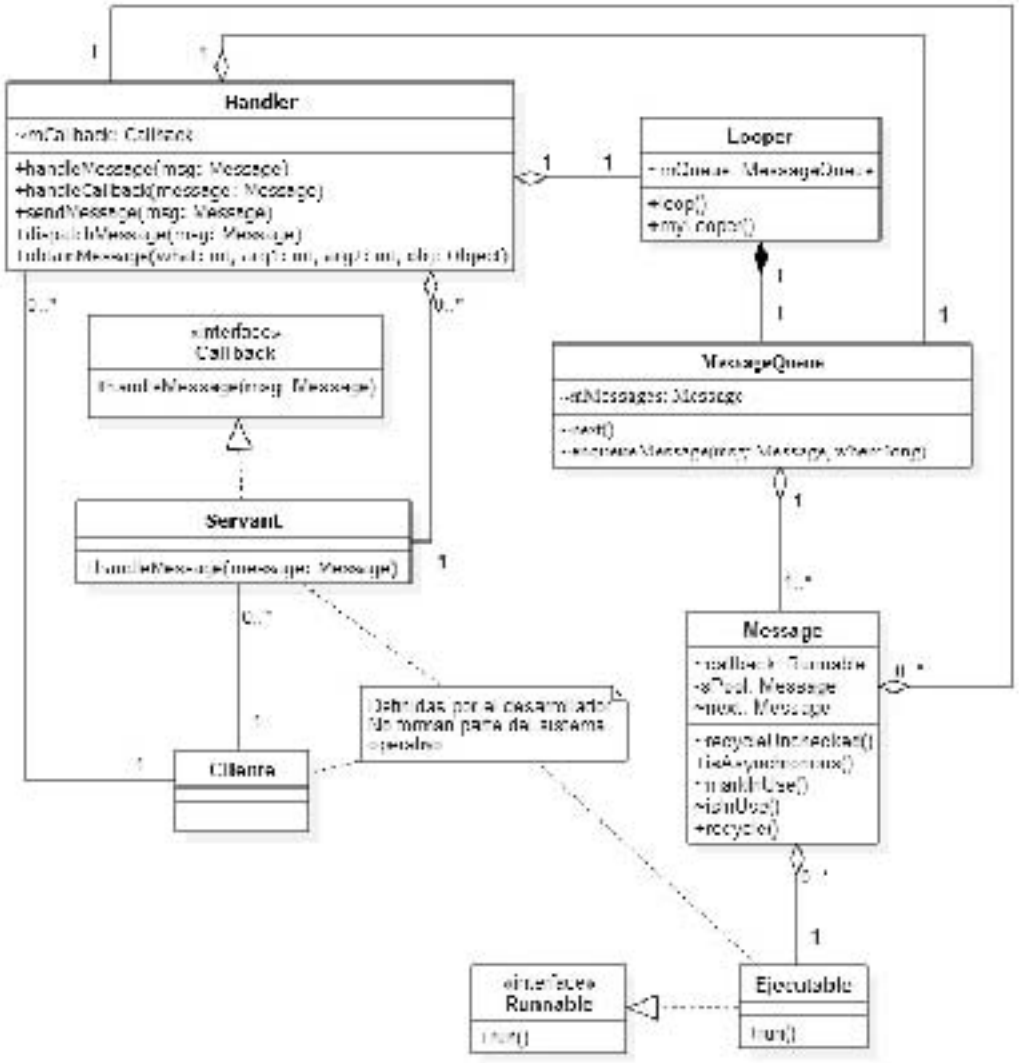

Figura 2.5: Diagrama de clases del patrón Objeto Activo implementado en Android 5.1.1.

Esta clase es el punto de entrada de la aplicación, por lo cual su instancia corre en el hilo primario de la misma. Nuevamente, de acuerdo a los diagramas, el siguiente objeto que se crea (aunque no es estrictamente necesario en este punto) es un Servant, que es el objeto que ejecuta la tarea solicitada. La clase Servant debe implementar la interfaz Callback, definida dentro de la clase Handler, pues en ella se define el método handleMessage(Message msg) que es a través del cual se reciben los mensajes (más adelante se muestra cómo). La interfaz solo incluye la definición de ese método:

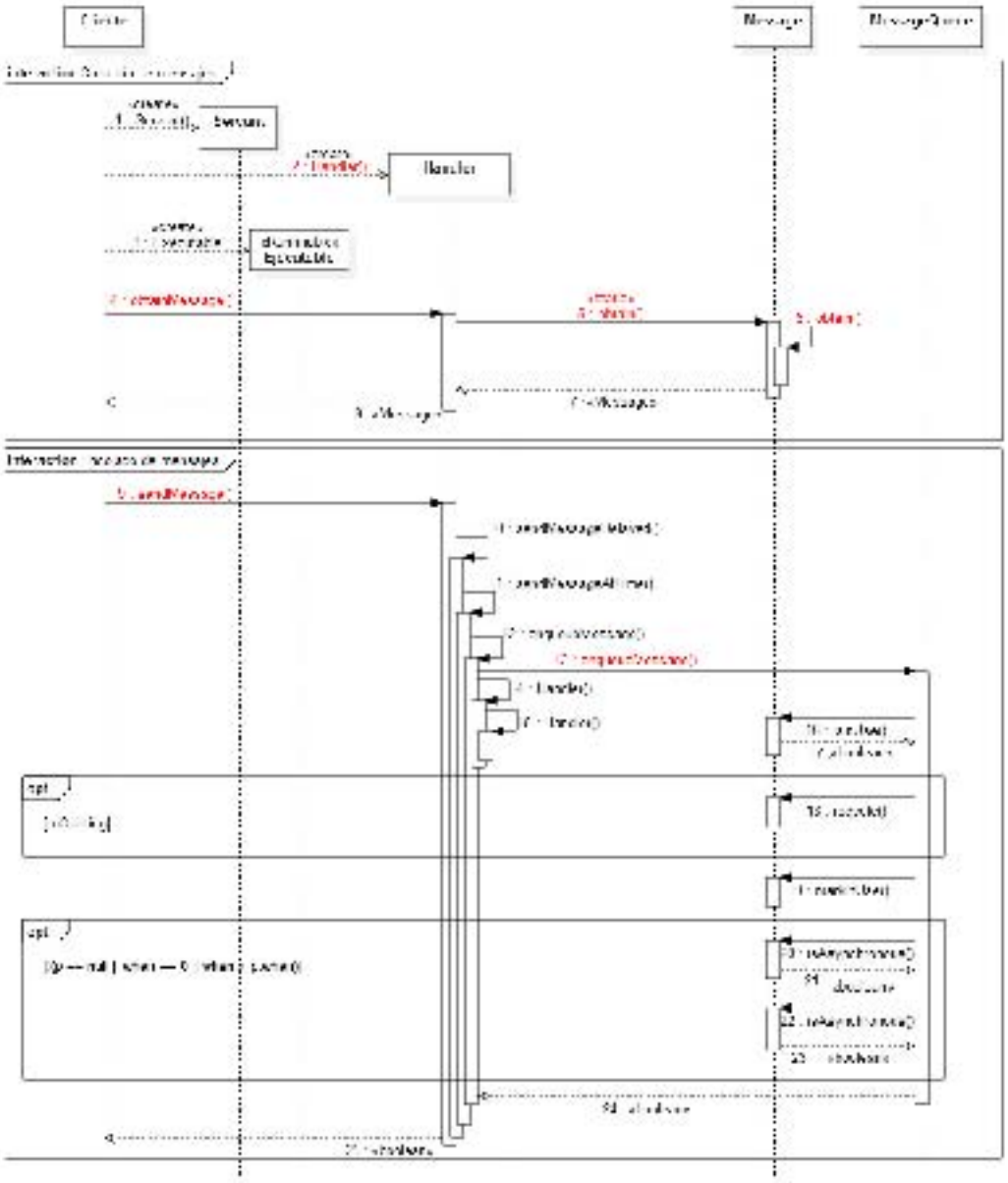

Figura 2.6: Diagrama de secuencia del patrón Objeto Activo en Android 5.1.1 (encolar mensajes).

```
public interface Callback {
   public boolean handleMessage(Message msg);
}
```

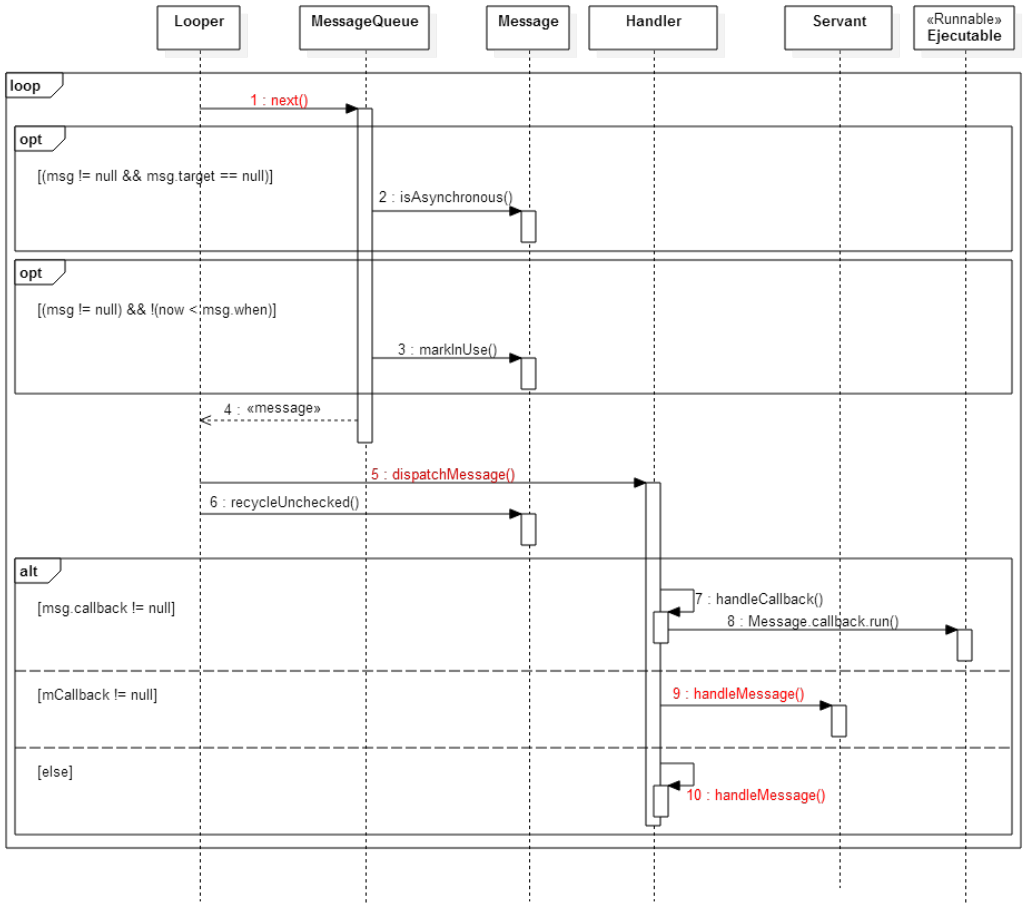

Figura 2.7: Diagrama de secuencia del patrón Objeto Activo en Android 5.1.1 (desencolar mensajes).

Para ejecutar un objeto de tipo Servant en un hilo distinto hay varias formas de hacerlo. Una de ellas es que la clase Servant implemente la Interfaz Runnable, con lo cual debe contener una definición del método run(). Esto ejecuta el objeto pasado como argumento en el nuevo hilo. Por ejemplo, se puede agregar un atributo de tipo Thread a la clase Servant, e instanciarlo en el constructor de la misma, pasándole como argumento una referencia a la misma instancia de Servant que se está creando, mediante la palabra reservada this. El código es el siguiente:

```
class Servant implements Handler.Callback, Runnable {
   //...
   Thread hilo;
```

```
   //...
   public Servant(){
      hilo = new Thread(this);
   }
   //...
}
```

Al instanciar la clase Thread no se ejecuta el hilo. Esto ocurre en el momento en que se llame a su método run(), y para que el planificador de hilos ejecute ese método, el hilo debe estar en estado de preparado, lo cual ocurre al ejecutar el método start(), por lo cual el constructor de la clase Servant solo crea su instancia y la de su atributo hilo.

Para indicarle al Servant el método que debe ejecutar, una opción es mediante los atributos del mensaje pasado como argumento, ya que como se observa en el código de la clase Message [15], ésta contiene un atributo de tipo entero llamado what. Cuando el Servant recibe el mensaje mediante el método handleMessage el valor del atributo what puede ser almacenado en un atributo de la clase Servant para utilizarlo al momento de la ejecución del método run(). Por lo tanto, una vez almacenado el valor de what, es posible invocar al método start() para iniciar la ejecución en otro hilo. En cuanto a la obtención del resultado por parte del cliente, existen varias formas de implementación. Por el momento, solo se debe tener en cuenta que el Servant escribe el resultado en un objeto "Future". El código de la clase Servant queda de la siguiente manera:

```
class Servant implements Handler.Callback, Runabble {
   int metodo;
   Thread hilo;

   public Servant(){
      hilo = new Thread(this);
   }

   @override
   public boolean handleMessage(Message message){
      metodo = message.what;
      hilo.start();
      return true;
   }
```

```
   @override
   public void run() {
      switch(metodo) {
         case 1:
            metodo1();
            break;
         case 2:
            metodo2();
break;
 //...
 case n:
    metodon();
break;
 default:
      }
   }

   private void metodo1() {
      //...
      //Escritura del resultado en un objeto Futuro
   }

   private void metodo2() {
      //...
      //Escritura del resultado en un objeto Futuro
   }
   //...
   private void metodon() {
      //...
      //Escritura del resultado en un objeto Futuro
   }
   //...
}
```

Una vez que se cuente con el código de la clase Servant, puede ser instanciada como cualquier otra clase en el punto que se necesite, por ejemplo:

```
Servant servant1;
servant1 = new Servant();
```

El siguiente objeto a crear es uno de la clase Handler, que se utiliza como Proxy para el envío de los mensajes al Servant, y al crearlo, se le envía este último para establecer el objeto destino de los mensajes:

```
Handler proxy1;
proxy1 = new Handler(servant1);
```

Como se observa en el siguiente código, cuando se pasa un objeto de tipo Callback como argumento al constructor, sucede lo siguiente: el constructor invoca una sobrecarga del mismo, pasándole como argumentos el objeto Callback y un valor false para establecer que el objeto no será asíncrono. Como se puede observar al final del constructor, el atributo mCallback apunta al objeto Callback pasado como argumento, y este atributo se utiliza para decidir en dónde se reciben los mensajes, ya que la clase Handler puede direccionar los mensajes a distintos lugares.

```
public class Handler{
   //...
   public Handler(Callback callback) {
      this(callback, false);
   }
   //...
   public Handler(Callback callback, boolean async) {
   if (FIND_POTENTIAL_LEAKS) {
      final Class<? extends Handler> klass = getClass();
      if ((klass.isAnonymousClass() || klass.isMemberClass()
          || klass.isLocalClass()) && (klass.getModifiers() &
          Modifier.STATIC) == 0) {
             Log.w(TAG, "The following Handler class should be
             static or leaks might occur: " +
             klass.getCanonicalName());
      }
   }

   mLooper = Looper.myLooper();
   if (mLooper == null) {
      throw new RuntimeException("Can't create handler " +
      "inside a thread that has not called Looper.prepare()");
   }
```

```
   mQueue = mLooper.mQueue;
   mCallback = callback;
   mAsynchronous = async;
   }
}
```

Ese direccionamiento se realiza en el método dispatchMessage(Message msg) como sigue:

```
public void dispatchMessage(Message msg) {
   if (msg.callback != null) {
      handleCallback(msg);
   } else {
      if (mCallback != null) {
         if (mCallback.handleMessage(msg)) {
            return;
         }
      }
      handleMessage(msg);
   }
}
```

El método dispatchMessage(Message msg), primeramente, verifica si el atributo callback (de tipo Runnable) del mensaje recibido es distinto de null. De ser así se invoca al método handleCallback(Message message) en donde se ejecuta su método run(), lo que significa que todos los mensajes que contengan un atributo callback distinto de null se ejecutan en un hilo independiente, pudiéndose crear una cantidad ilimitada de ellos. El método handleCallback está definido como se muestra a continuación, al ejecutarse la instrucción message.callback.run();, el método run() se ejecuta de principio a fin, sin alternar el uso del procesador con otros hilos:

```
private static void handleCallback(Message message) {
   message.callback.run();
}
```

Al ejecutarse la instrucción message.callback.run();, el método run() se ejecuta de principio a fin, sin alternar el uso del procesador con otros hilos.

Si el atributo callback del mensaje es nulo, se verifica el atributo mCallback de la clase Handler (de tipo Callback) y de ser distinto de nulo se invoca su método handleMessage(Message message) en donde se realiza cualquier

procesamiento con el mensaje. Es por esto que al crear la instancia de la clase Handler se puede pasar como argumento un objeto de tipo Callback como el objeto servant1 mostrado anteriormente.

Y por último, si el atributo mCalback es nulo, se invoca al método handleMessage(Message message) de la misma clase Handler, fungiendo como Proxy y Servant al mismo tiempo, ya que en algún lugar se deben recibir y manejar los mensajes, esto ocasiona que la invocación y ejecución de los métodos se realicen en el mismo hilo, incumpliendo con el objetivo del patrón de desacoplar esas acciones al realizarlas en distintos hilos.

Ahora bien, quien envía los mensajes al método dispatchMessage(Message msg) de Handler es la clase Looper, específicamente, mediante el método loop(), que como lo muestra el código siguiente, utiliza un ciclo for sin condición de parada para desencolar los mensajes de la variable queue de tipo MessageQueue. Ésta que es la Activation List asociada con el Scheduler ó Looper. Al utilizar un for sin condición de parada, el método no termina hasta que la cola está vacía, es decir, cuando la condición if(msg == null) es verdadera, y se ejecuta return. Sin embargo, el método puede ser implementado de tal manera que se utilice la clase MessageQueue como una cola de prioridades, modificando el orden de ejecución. Para lo cual se deben realizar validaciones previas a la ejecución, verificando los atributos de los objetos Message o ejecutando algunos métodos Guard que se pueden agregar a la definición de la clase. Esto puede ser utilizado por el desarrollador en casos específicos para mejorar la responsividad.

```
public static void loop() {
   final Looper me = myLooper();
   //...
   for (;;) {
      Message msg = queue.next(); // podría bloquear
      if (msg == null) {
      //No message indicates that the message queue is quitting
         return;
      }
      //...
      msg.target.dispatchMessage(msg);
      //...
   }
}
```

Cada objeto Message está relacionado con una sola instancia de Handler y la referencia a esa instancia se guarda en el atributo target al momento de su creación. Al momento de desencolarlos se utiliza este atributo para invocar al método dispatchMessage(Message msg) de la instancia adecuada. Cabe mencionar que cada instancia de MessageQueue está asociada con una instancia de Looper, lo que limita el proceso de desencolado de las peticiones.

Hasta este punto se sabe que se cuenta con un objeto de tipo Handler que es quien recibe los mensajes dirigidos a un objeto de tipo Servant. Éstas instancias de la clase Message que están relacionadas con el Handler se encolan en una instancia de la clase MessageQueue, y son desencolados por el método estático de una instancia de Looper. Nuevamente las dos relacionadas con una sola instancia de Handler. Pero, ¿en dónde se crean y relacionan estas instancias?, pues bien, como se sabe la clase Handler tiene varios constructores como los ya descritos, y al invocar el constructor que recibe como argumento un objeto de tipo Callback, éste invoca otro constructor enviándole el mismo argumento mas un valor boolean. En ese constructor el atributo mLooper de la clase Handler se iguala con un objeto de tipo Looper, que se obtiene por medio del método estático myLooper() de la clase Looper:

```
public Handler(Callback callback, boolean async) {
   //...
   mLooper = myLooper();
   //...
}
```

Esto quiere decir que al momento de crear una instancia de Handler ya existe una instancia de Looper y es ésa la que se utiliza, ¿quién crea previamente esa instancia? Al momento de ejecutar una aplicación en Android, el entorno de Android ejecuta en automático el método prepareMainLooper() y dentro de éste se invoca al método prepare(boolean quitAllowed) como se muestra a continuación. Cuando se ejecuta sThreadLocal.set(new Looper(quitAllowed)), se crea un objeto de tipo Looper y se almacena en el atributo sThreadLocal. Es ése objeto Looper el que devuelve el método myLooper, aunque existe la alternativa de crear un nuevo objeto y pasarlo como argumento en un constructor de Handler.

```
public final class Looper {
   //...
```

```
   public static void prepareMainLooper() {
      prepare(false);
      synchronized (Looper.class) {
         if (sMainLooper != null) {
            throw new IllegalStateException("The main Looper" +
            " has already been prepared.");
         }
         sMainLooper = myLooper();
      }
   }
   private static void prepare(boolean quitAllowed) {
      if (sThreadLocal.get() != null) {
         throw new RuntimeException("Only one Looper may be" +
         " created per thread");
      }
      sThreadLocal.set(new Looper(quitAllowed));
   }
   //...
}
```

Con respecto a la Activation List, al crear la instancia de Looper con el constructor invocado en sThreadLocal.set(new Looper(quitAllowed)) también se crea una instancia de la clase MessageQueue, como se observa en el código del constructor que se muestra a continuación. Ésa instancia se guarda en el atributo mQueue de la clase Looper, el cual es final, y por lo tanto, cada objeto Looper está asociado con una sola Activation List mQueue.

```
private Looper(boolean quitAllowed) {
   mQueue = new MessageQueue(quitAllowed);
   mThread = Thread.currentThread();
}
```

Finalmente, para enviar mensajes al Servant, se deben especificar las operaciones solicitadas en objetos de tipo Message. Estos objetos se obtienen a través de la clase Handler, específicamente con el método obtainMessage(), y a través de su atributo what (de tipo int) se indica la operación solicitada. La forma de hacerlo es la siguiente:

```
Message msg;
msg = proxy1.obtainMessage();
msg.what = 1;
```

Al invocar al método obtainMessage() de la clase Handler, éste invoca al método estático obtain(Handler h) de la clase Message en donde se establece el Proxy asociado al mensaje mediante el atributo target:

```
public class Handler {
   //...
   public final Message obtainMessage()
   {
      return Message.obtain(this);
   }
   //...
}

public final class Message implements Parcelable {
   //...
   public static Message obtain(Handler h){
      Message m = obtain();
      m.target = h;
      return m;
   }
   //...
}
```

Al contar con los objetos de tipo Message, simplemente se invoca al método sendMessage(Message msg) de la clase Handler para encolarlos:

```
representate.sendMessage(msg);
```

De acuerdo a lo descrito, un cliente solicita instancias de la clase Message a través de un objeto Handler. En el caso de una aplicación este cliente se ejecuta en algún hilo de la misma y no es parte del sistema operativo, al igual que la instancia de Handler. Una vez que se ha establecido el estado de las instancias tipo Message, éstas se envían a través de la misma instancia de Handler a la actual instancia de la clase MessageQueue, utilizada por el sistema operativo que funge como Activation List, mediante el método sendMessage(Message msg). A continuación una instancia del Scheduler del patrón implementado mediante la clase Looper, que se encuentra corriendo en el sistema operativo desde que es cargado, desencola las Method Requests representadas o almacenadas en los objetos Message y los envía de nuevo a la instancia de Handler mediante la cual fueron encoladas. Esto a través del método dispatchMessage(Message msg). En este método, se decide cómo tratar al mensaje recibido de acuerdo a su atributo de tipo Runnable, callback,

y al atributo de tipo interfaz Callback (clase que implementa la interfaz), mCallback de la clase Handler. Si el atributo callback del mensaje no es nulo, se ejecuta de principio a fin mediante su método run(); si es nulo y el atributo mCallback de Handler no lo es, se le envía el mensaje mediante el método handleMessage(Message msg) para que lo procese como lo haya definido el desarrollador. En el código descrito, por ejemplo, se utiliza su atributo what para identificar que método se solicita y ejecutarlo en un nuevo hilo creado en la instancia de Servant; y si el atributo mCallback también es nulo, el mensaje se envía al método handleMessage(Message msg) de la misma instancia de Handler, nuevamente para ser procesado como lo haya definido el desarrollador. En el código descrito, los mensajes se envían a las correspondientes instancias de Servant, ya que ése es el valor que adquiere el atributo mCallback del objeto Proxy de tipo Handler al crearlo, pasándole un objeto de tipo Servant en el constructor. Este proceso se describe de manera gráfica en la figura 2.8.

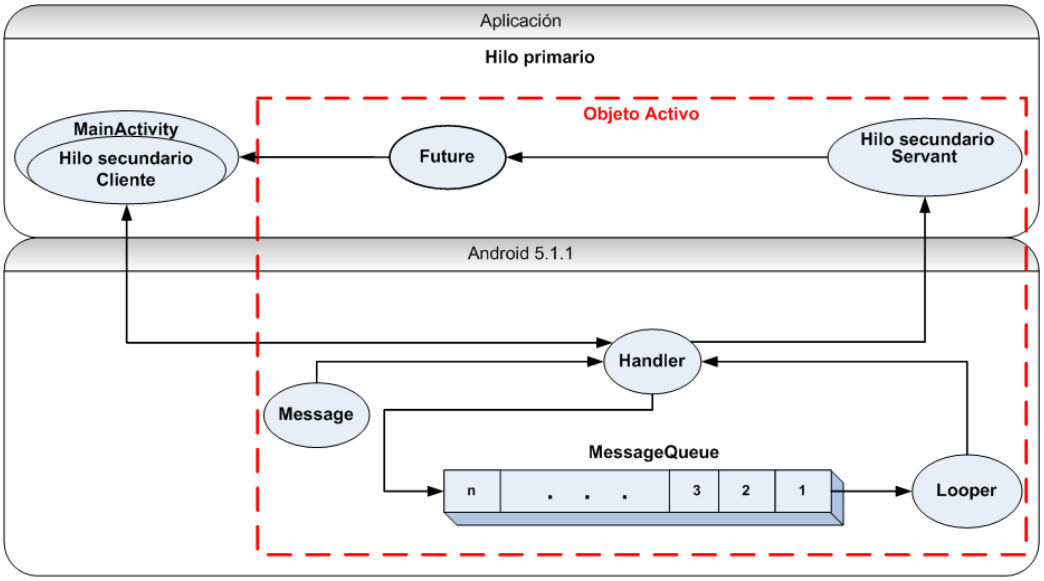

Figura 2.8: Esquema de uso de Objeto Activo en Android 5.1.1.

2.4. Métricas de software

Cuando se trata de comprender mejor los atributos de los modelos que se crean y evaluar la calidad de los productos de la ingeniería o de los sistemas que se construyen, la medición es un elemento clave. Pero a diferencia de otras disciplinas de la ingeniería, la Ingeniería de Software no se basa en las

leyes cuantitativas básicas de la física. Las medidas directas, como el voltaje, la masa, la velocidad o la temperatura no son comunes en el mundo del software. Debido a que las medidas y métricas del software suelen ser indirectas están expuestas al debate [17].

Aunque medida, medición y métrica son términos que suelen utilizarse de manera intercambiable, es importante observar las sutiles diferencias entre ellos. En el contexto de la Ingeniería de Software una **medida** proporciona una indicación cuantitativa de la extensión; la cantidad, la dimensión, la capacidad o el tamaño de algún atributo de un producto o proceso. **Medición** es el acto de determinar una medida. El Glosario de estándares del IEEE define **métrica** como una "medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo determinado" [17][20].

Cuando se ha recopilado un solo tipo de datos, se ha establecido una medida. La medición ocurre como resultado de la recopilación de uno o más puntos de datos (por ejemplo, se investigan varias revisiones de componentes y pruebas de unidad para reunir medidas del número de errores encontrados en cada uno). Una métrica de software relaciona de alguna manera las medidas individuales (por ejemplo, el número promedio de errores encontrados en cada revisión o prueba de unidad) [17].

Un ingeniero de software recopila medidas y desarrolla métricas para obtener los indicadores. Un **indicador** es una métrica o una combinación de métricas que proporcionan conocimientos acerca del proceso del software, un proyecto de software o el propio producto. Un indicador proporciona conocimientos que permiten al jefe de proyecto o los ingenieros de software ajustar el proceso, el proyecto o el producto para que las cosas mejoren [17].

Pues bien, para evaluar si el desempeño de la propuesta mejora, se necesita de alguna herramienta basada en la medición de los tiempos de ejecución. Las herramientas que aquí se proponen para comprobar si la hipótesis de ésta tesis es cierta, son las siguientes métricas:

- El tiempo promedio de ejecución, definido como el tiempo total entre el comienzo y la finalización de un evento (o proceso, en este caso el envío, encolamiento, procesamiento y retorno del resultado de una cantidad de Method Requests).
- El desempeño, que se define como el monto total de trabajo realizado en un tiempo dado.

Ya que las muestras utilizadas son pequeñas (menos de 30 observaciones), para el tiempo promedio de ejecución se utiliza el estadístico t de Student, y como es común en estos estudios, con un 95 % de confianza. La fórmula que expresa la métrica es la siguiente:

$$\mu = \bar{x} \pm t\frac{S}{\sqrt{n}} \tag{2.1}$$

Donde:
μ es el tiempo promedio de ejecución,
$\bar{x}$ es el promedio de la muestra,
t es la constante del estadístico t de Student,
S es la desviación estándar de la muestra, y
n es el tamaño de la muestra

En cuanto a la segunda métrica, que se utiliza para observar la relación entre la cantidad de peticiones conocidas como Method Requests que atiende el patrón y el tiempo utilizado, la fórmula es la siguiente:

$$d = \frac{p}{t} \tag{2.2}$$

Donde:
d es el desempeño,
p es la cantidad de Method Requests y,
t es el tiempo

2.5. Resumen

En el presente capítulo se introducen los conceptos y definiciones básicas utilizadas en el presente trabajo. Primeramente, se introduce el concepto de Patrón de Software, y la forma POSA, que es una de las formas utilizadas en la comunidad de la Ingeniería de Software para describir los Patrones de Software.

A continuación, se presenta el patrón específico en que se basa la tesis: el Patrón Objeto Activo. Éste patrón, se utiliza cuando se desea desacoplar la invocación de los métodos necesarios de su ejecución. El patrón contiene algunos elementos para recibir las peticiones de ejecución de método o Method Requests de parte de código cliente, mantenerlos en una cola de espera,

planificar su ejecución y finalmente ejecutarlos para retornar el resultado al código cliente. La invocación y la ejecución de los distintos métodos se realiza en hilos distintos.

El siguiente tema que contiene el capítulo, es una descripción general del sistema operativo Android y su arquitectura interna, principalmente para aclarar qué parte de Android contiene los elementos de código involucrados en el patrón que es analizado. En la siguiente sección, se describe en detalle el código que forma el patrón Objeto Activo en Android, la manera en que se utiliza y algunas de sus deficiencias.

Y como último punto, se presenta la definición de Métrica de Software, y se proponen dos métricas específicas necesarias para el análisis de los datos obtenidos del experimento: el tiempo promedio de ejecución y el desempeño.

Capítulo 3

Trabajo relacionado

Con el creciente uso de dispositivos móviles, se ha incrementado el interés en conocer el desempeño de los mismos y como medirlo, por lo que se ha realizado bastante investigación en torno al tema. Sin embargo, las publicaciones en torno a los patrones utilizados en programación concurrente en Android son muy escasas. Aquí se presentan algunos trabajos directamente relacionados con el tema de esta tesis, aunque no aborden específicamente el tema de la implementación en Android de algún patrón para el manejo de concurrencia y/o mejora del desempeño. Abordan el concepto o patrón de Objeto Activo, tema básico utilizado para conseguir el objetivo del presente trabajo. Los trabajos descritos son los siguientes:

- Resource Management for Mobile Operating Systems based on the Active Object Model [13].
- Pattern-Based Simulation: Simulating the Actor Model Using the Active Object Behavioural Pattern [14].

3.1. Resource Management for Mobile Operating Systems based on the Active Object Model

Para establecer un contexto del cual partir, el resumen del trabajo señala el hecho de que los dispositivos personales están penetrando en la mayoría de los sectores del mercado de tecnología. Cada vez es más común que estos dispositivos contengan en alguna capa de software un sistema operativo móvil embebido. En la última década, los dispositivos móviles han llegado a ser

parte de algunos dominios de tiempo real. Sin embargo, los autores afirman que la adopción total de los dispositivos personales embebidos en entornos de tiempo real aún no ha sido posible, debido a su falta de predictibilidad, que se deriva entre otras cosas, del modelo de concurrencia y ejecución del sistema operativo.

A partir de tal hecho, se establece que se necesitan mecanismos para la administración de recursos eficiente y oportunamente, para conocer, por lo menos, las restricciones de los dominios de aplicación emergentes que son altos consumidores de recursos. Como respuesta a esa necesidad, se presenta un esquema para integrar las técnicas de administración de recursos con el modelo de concurrencia de los sistemas operativos embebidos, que usan el modelo de concurrencia de Objeto Activo utilizando como ejemplo el sistema operativo Symbian.

En la sección de Antecedentes y trabajo relacionado, los autores mencionan que la efectividad de la administración de los recursos en los dispositivos móviles depende entre otras cosas, de las características de la plataforma de hardware, así como de las primitivas y el modelo de concurrencia que provea el sistema operativo. Partiendo de tal aseveración, es que abordan el tema del modelo de concurrencia basado en Objetos Activos en la sección titulada Soporte de Ejecución en los Sistemas Operativos Móviles, en el cual, primeramente, se describe a grandes rasgos el funcionamiento del modelo y posteriormente, algunos problemas relacionados con el mismo. La esencia de esta sección se resume a continuación.

El Modelo de Objeto Activo (AOM por sus siglas en inglés) es un framework que sigue el Principio Abierto Cerrado (OCP por sus siglas en inglés) que indica que los módulos deberían ser diseñados de tal manera que después puedan ser extendidos pero no modificados. El propósito es que el programador pueda usar el framework para la mayoría de su trabajo con solo extenderlo.

En éste modelo, la declaración de un Objeto Activo encapsula las peticiones a un proveedor de servicio asíncrono y el manejo de las peticiones terminadas. Por cada objeto activo que ofrece su propia funcionalidad, debería existir un hilo en código. El modelo contiene una entidad de Active Scheduler que determina el orden de ejecución de los Objetos Activos, de tal manera que cuando exista una petición de ejecución de uno de ellos, su método RunL() se invoca siempre que no existan peticiones pendientes de otros objetos con mayor prioridad. La entidad Active Scheduler puede tener

uno o más Objetos Activos que se atienden dependiendo de su prioridad y peticiones, como se muestra en la figura 3.1. Es necesario que solo exista un Scheduler (Planificador) por hilo. Esta instancia maneja todos los Objetos Activos de ese hilo.

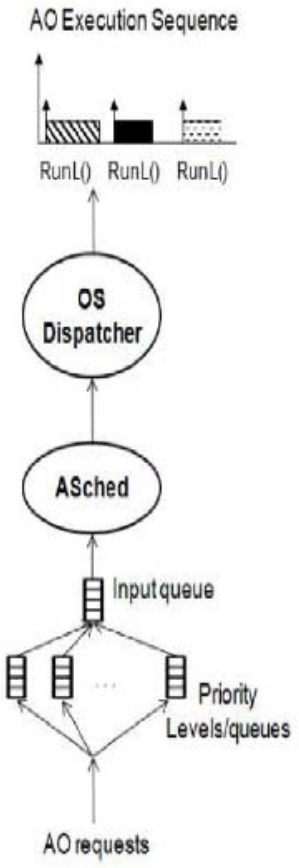

Figura 3.1: Diagrama de ejecución de AOM [13].

Hasta este punto de la sección, se describe el modelo de concurrencia basado en Objetos Activos, y como es manejado en Symbian. Básicamente es la descripción de la implementación del patrón Objeto Activo en ese sistema operativo móvil. Esa descripción es la principal similitud con esta tesis, al mismo tiempo que presenta una diferencia importante: es una implementación en Symbian, no en Android. En seguida, se describe un problema relacionado a los retardos de ejecución en los Objetos Activos en Symbian, mencionando el código involucrado y su funcionamiento, detalles muy específicos de ese sistema.

Para poder manejar los retardos mencionados, se presenta el patrón A/Th, que extrae los beneficios tanto del Modelo de Objeto Activo o AOM, como del modelo de hilos: en él se puede integrar un análisis de tiempo real en tiempo de diseño para calcular el retardo de la ejecución de la devolución de llamada, y cuantificar tal retardo en tiempo de ejecución dentro del hilo o función RunL().

Para calcular el retardo que se menciona, se presenta un par de fórmulas muy sencillas, que se usan para realizar un análisis de retardo en tiempo de diseño:

- Tiempo de ejecución del Objeto Activo i sin retardos: $C_i = C_i$
- Tiempo de ejecución del Objeto Activo i con retardos: $C_i = C_i + \delta_i$ Donde δ_i es el retardo del Objeto Activo i

En cuanto al modelo de ejecución A/Th, es la mejor opción para aplicaciones de tiempo real, por ejemplo cuando existen varias actividades que deben ser desempeñadas en paralelo esto solo puede ser hecho en un contexto multihilo. En el texto se comenta que puede ser lógicamente inferido que el grado de ejecución concurrente es comparable al modelo de hilos puro.

AOM sigue una invocación secuencial de la funcionalidad de acuerdo a las prioridades asignadas a los Objetos Activos, por lo tanto solo un hilo es percibido por el programador, y las devoluciones de llamada de los objetos que hayan sido invocados más tardíamente en una secuencia de invocaciones serán las más afectadas por los retardos.

Después de presentar las ventajas y desventajas de los modelos de concurrencia basados en Objetos Activos e hilos, se presenta la principal aportación del trabajo: Un prototipo de arquitectura de administración de recursos. Para la propuesta, se aclara que se consideraron como recursos el procesador y la memoria solamente. En esta propuesta, se añadió un objeto activo adicional como parte del sistema operativo que monitorea la ejecución del sistema con respecto a la utilización del procesador y la memoria.

Según el texto, las facilidades de administración de los recursos con la propuesta son:

- Contabilización de los recursos. Que está soportada por la función MonitorUsage() y sirve para determinar si hay sub utilización o sobre utilización de recursos.

- Cumplimiento de uso de los recursos. Esta funcionalidad recae en la utilización de los temporizadores del sistema operativo y no permite que los hilos u objetos utilicen más tiempo del procesador que el acordado.
- Lógica de Planeación. Está basada en la monitorización de los objetos, y determina la secuencia de ejecución de acuerdo a varios patrones, aunque no se menciona cuáles.

La arquitectura introduce un hilo que actúa como administrador de los recursos Llamado TTActiveManager (Time Triggered Active Manager) que se basa o auxilia de algunas entidades auxiliares:

- ReqCount. Provee del conteo de peticiones y realiza el conteo del uso de recursos.
- ReqSched. Ejecuta el análisis de planeación y realiza un cálculo de ejecución cíclica para determinar el grado de cumplimiento de los límites de tiempo.
- ResTrade. Realiza el cálculo en el siguiente periodo de ejecución.

El fucionamiento de la propuesta es el siguiente: El Active Scheduler ASched usa la entidad TTActiveManager para manejar la activación de los objetos activos. Por cada activación, se realiza el conteo por medio de la entidad ReqCount después del análisis de planeación realizado por la entidad ReqSched. La entidad Restrade puede realizar diferentes cálculos para las activaciones del siguiente periodo, esto puede ser llevado a cabo analizando los datos recabados por ReqCount. Los autores aclaran que en el framework actual, la funcionalidad de ResTrade es subespecificado, lo que muestra un framework conservador, pero de cualquier manera se garantiza un comportamiento de tiempo real. Para lograr previsión temporal, se introdujo algo de sobrecarga debido a la funcionalidad extra de las clases ReqSched, ReqCount, y ResTrade.

Para obtener mediciones, el esquema de administración de recursos se emuló e integró en pruebas específicamente para procesador y memoria. La arquitectura muestra un uso de los recursos uniforme desde un punto de vista de observador y en las pruebas no se muestra alto riesgo de indeterminismo o picos de uso de recursos.

Únicamente cabe mencionar que en el caso de usar multihilos, el uso de la memoria se incrementa debido a que cada hilo tiene su propia pila. En el

caso de AOM, como no hay necesidad de exclusión mutua para salvaguardar los accesos concurrentes, el desempeño se incrementa y el código se simplifica. Los autores aseguran que usando el patrón propuesto con objetos activos dentro de un hilo es más rápido que simplemente usar un modelo de objeto activo, ya que en él no es obligatorio el uso de un Active Scheduler y la activación de los Objetos Activos es más rápida.

Finalmente, se emuló la arquitectura de administración de recursos por medio de una implementación sintética para comparar los distintos modelos de ejecución analizados en el trabajo. Los resultados muestran que el intercambio de contexto repercutió en el desempeño del modelo de ejecución basado en Hilos y en A/Th. Con respecto a AOM, no se registró retardo y además, se comenta que la simplicidad de su modelo de API compensa la pequeña sobrecarga que se da por la activación de los objetos y que esta puede ser considerada insignificante.

A diferencia del objetivo y aportaciones de la presente tesis, este trabajo se enfoca, primeramente, en desarrollar un modelo de concurrencia alternativo al uso de ejecución basada en hilos, con un paradigma de Objeto Activo, llamado A/Th, y posteriormente, lo utilizan para construir una arquitectura de administración de recursos, que aprovecha las facilidades introducidas por los modelos de hilos y Objetos Activos.

3.2. Pattern-Based Simulation: Simulating the Actor Model Using the Active Object Behavioural Pattern

Se propone el uso de patrones de software como herramientas de base viables para simular y analizar atributos deseables de sistemas de software específicos. Concretamente, se propone el uso del patrón Objeto Activo en combinación con técnicas de simulación estocástica para producir un modelo de simulación. El modelo de simulación presentado recibe como entrada el patrón de comportamiento de un Objeto Activo y elementos de teoría de colas, y produce como salida estimaciones acerca del comportamiento de desempeño del Objeto Activo.

Esta descripción pone de manifiesto que el objetivo del trabajo es diferente al de la presente tesis, y al mismo tiempo muestra la relación: Se presenta un modelo de simulación que realiza estimaciones acerca del comportamiento

del desempeño de un Objeto Activo, mientras que en esta tesis se modifica el código de los componentes que conforman el patrón Objeto Activo en Android, en busca de que mejore su desempeño en ese sistema operativo.

Primero, se describe el Modelo de Actor (u Objeto Activo), que se desarrolló como una extensión a la programación Orientada a Objetos, en busca de eliminar el límite impuesto al considerar la programación como una secuencia de acciones. Comúnmente los lenguajes de programación orientados a objetos son secuenciales debido a que solo permiten a un objeto estar activo en un momento preciso durante la ejecución de la programación.

Basados en el modelo de actor, los actores se proponen como los bloques de construcción subyacentes básicos de la programación concurrente. Una aplicación de actor consiste en una colección de objetos asíncronos que se ejecutan concurrentemente. Los actores son objetos autónomos y concurrentes, ejecutando a su propio ritmo, y son capaces de comunicarse enviándose mensajes entre sí [21] [22].

Para que los Objetos Activos actúen de tal forma, éstos poseen cierta estructura, y los elementos que la forman son los siguientes:

- Mecanismo de entrega de mensajes. Se utiliza como interfaz para recibir los mensajes por parte del actor.
- Cola de entrada. Almacena los mensajes recibidos por el actor hasta que puedan ser despachados.
- Mecanismo de planeación. Define el plan de ejecución de los mensajes. En el original Modelo de Actor el mecanismo es simple: Los mensajes se ejecutan en el orden de llegada por el comportamiento actual. Las variaciones del mecanismo de planeación permiten que los mensajes en la cola de entrada se ejecuten en base a un criterio distinto que el orden de llegada.
- Métodos. Son los diferentes comportamientos definidos para las respuestas a los mensajes de un actor.
- Estado. Es el conjunto de valores de las variables de la instancia que colectivamente representan el estado de un actor.

La figura 3.2 muestra la relación entre los elementos del actor.

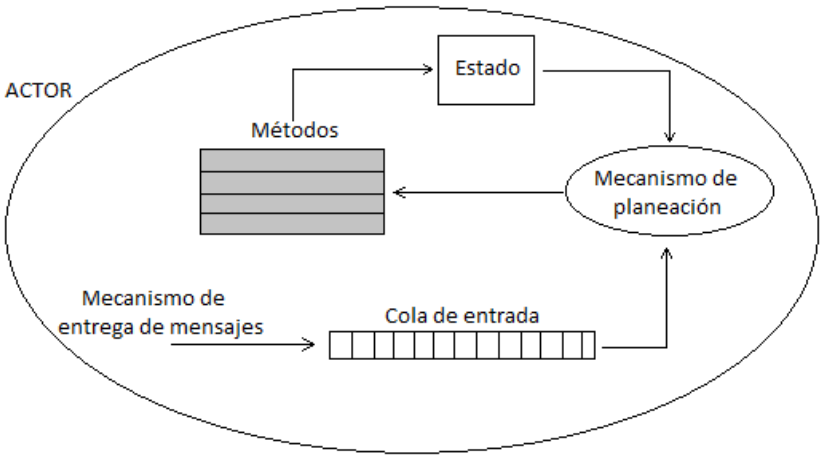

Figura 3.2: Estructura de un actor [14].

Para el modelo de simulación, los tiempos entre las llegadas de las peticiones, se modelan mediante el proceso de llegada de Poisson, en el que los eventos son tomados de una larga población, en donde cada miembro es independiente de los demás. Con este proceso, es posible calcular la probabilidad de que n peticiones se reciban en un intervalo de tiempo, utilizando una tasa media de llegada.

Para los tiempos de servicio, se considera la misma propiedad Markoviana que para los tiempos de llegada, es decir, que ningún evento depende de los eventos pasados o futuros. Al igual que para los tiempos de llegada, se proporciona una función para calcular la probabilidad de atender una cantidad μ de servicios en un ntervalo de tiempo.

Sabiendo que la llegada de peticiones forma un proceso de Poisson con una tasa de llegada de mensajes promedio por segundo λ, y el tiempo de procesamiento por mensaje es una variable aleatoria distribuida exponencialmente con un tiempo de servicio de mensajes por segundo μ, se plantea el objetivo de contestar las siguientes preguntas:

- ¿Cuánto tiempo puede tardar un objeto activo procesando un número de mensajes?
- ¿Cuánto tiempo puede el actor estar ocioso?
- ¿Cuál es el tiempo de respuesta promedio observado en que las peticiones son manejadas por el objeto activo?

Para responder estas preguntas, se utiliza una ecuación obtenida de [23], que calcula la probabilidad de que un objeto activo esté en el estado E en el instante t, es decir mientras tiene n mensajes en servicio o esperando por servicio.

Acerca del modelo de simulación, un punto importante que se considera de una simulación de eventos discretos de una estructura de cola, es generar una variable aleatoria de una distribución arbitraria. Debido a la naturaleza de los tiempos entre llegadas y servicio de peticiones, es importante generar variables aleatorias distribuidas exponencialmente.

El siguiente punto del trabajo es la implementación del modelo, en donde se presenta incluso parte del código en C++. A continuación, se presenta el experimento realizado con medidas de una aplicación real de Objeto Activo como parámetros de entrada para el modelo de simulación.

Por medio de la contribución de ese trabajo, se establece una relación con la presente tesis adicional al patrón utilizado, ya que los tiempos medidos en la ejecución del código propuesto se pueden utilizar como datos de entrada para ese modelo de simulación, para prever su comportamiento.

Los resultados de la experimentación, indican que los tiempos de procesamiento y comunicación calculados por el modelo son muy próximos a los tiempos reales, mientras que el tiempo de ocio presenta un comportamiento errático. Al comparar los tiempos totales, se observa que los tiempos totales estimados y los tiempos totales reales, obtenidos de la medición de la ejecución del programa real, son parecidos aunque no exactos. Sin embargo, los datos que proporciona el modelo, están relacionados con el desempeño promedio del sistema.

3.3. Resumen

Con el creciente uso de los dispositivos móviles en dominios de tiempo real, que contienen un sistema operativo embebido en alguna capa de software, el primer trabajo relacionado: Resource Management for Mobile Operating Systems based on the Active Object Model, se enfoca en proveer un mecanismo para la administración de recursos eficiente y oportunamente, para conocer, por lo menos, las restricciones de los dominios de aplicación emergentes que son altos consumidores de recursos de acuerdo a los propios autores. En respuesta a esa necesidad, se presenta un esquema para integrar

las técnicas de administración de recursos con el modelo de concurrencia de los sistemas operativos embebidos, que usan el modelo de Objeto Activo, usando como ejemplo el sistema operativo Symbian.

Primero se describen el Modelo de Objeto Activo, y algunos problemas de retardo en tiempo de ejecución relacionados el mismo, en su implementación en Symbian. En seguida, se presenta el patrón A/Th, que contiene los beneficios del Modelo de Objeto Activo y del modelo de hilos, que también se utiliza en la ejecución de tareas concurrentes, en él se puede integrar un análisis de tiempo real en tiempo de diseño para calcular el retardo de la ejecución de la devolución de llamada, y cuantificar tal retardo en tiempo de ejecución dentro del hilo.

Posteriormente, se utiliza el patrón A/Th para crear una arquitectura de administración de recursos, que permite la asignación adecuada de los recursos de la plataforma a través de las diferentes unidades de ejecución (hilos, tareas, objetos activos, etc.), de una manera que pueden desempeñar su funcionalidad cumpliendo con las restricciones temporales de haber alguna.

Al final, se emuló la arquitectura de administración de recursos por medio de una implementación sintética para comparar los distintos modelos de ejecución analizados en el trabajo. Los resultados muestran que el intercambio de contexto repercutió en el desempeño del modelo de ejecución basado en Hilos y en A/Th. Con respecto a AOM, no se registró retardo y, además, se comenta que la simplicidad de su modelo de API compensa la pequeña sobrecarga que se da por la activación de los objetos y que esta puede ser considerada insignificante.

El segundo trabajo revisado: Pattern-Based Simulation: Simulating the Actor Model Using the Active Object Behavioural Pattern, propone el uso del patrón Objeto Activo en combinación con técnicas de simulación estocástica, para construir un modelo de simulación que produce estimaciones acerca del comportamiento del desempeño de un Objeto Activo.

El trabajo comienza con la descripción del Modelo de Actor, que se desarrolla como una extensión a la tradicional programación orientada a objetos en busca de eliminar el límite impuesto al considerar la programación como una secuencia de acciones. Estos actores se proponen como los bloques de construcción subyacentes básicos de la programación concurrente.

Para el modelo de simulación, se considera que los intervalos de tiempo

entre llegada de las peticiones que recibe el actor, así como los tiempos de servicio o procesamiento poseen la propiedad de ser un proceso Markoviano, es decir que ningún evento depende de los eventos anteriores o posteriores. Para simular los tiempos de llegada, espera y procesamiento de las peticiones, se determinan las distribuciones de probabilidad adecuadas.

Se presenta la interfaz (métodos públicos) de la clase programada en lenguaje C++ para implementar el modelo de simulación, y se utilizan los tiempos medidos de una aplicación de Objeto Activo como argumentos de entrada del modelo en la experimentación realizada.

Los resultados de la experimentación, indican que los tiempos de procesamiento y comunicación calculados por el modelo son muy próximos a los tiempos reales, mientras que el tiempo de ocio presenta un comportamiento errático.

Capítulo 4

Análisis del desempeño de los dispositivos Android derivado de diversas implementaciones del patrón Objeto Activo.

Como alternativa para incrementar el desempeño del sistema, liberando al programador de la responsabilidad de los hilos Servants, es posible implementar el patrón Objeto Activo como una de sus variantes mencionadas en [4]: Objeto Activo Multi-hilos. En este capítulo se presentan los cambios necesarios en el código original de Android, así como una breve descripción de su uso.

4.1. Modificaciones propuestas para el patrón Objeto Activo en el sistema operativo Android: Objeto Activo Multi-hilos.

Para obtener esta variante, el patrón Objeto Activo debe incluir un grupo de hilos (o thread pool como se le conoce en el idioma inglés). El *Scheduler* debe ser configurado para asignar las Method Requests a los hilos del grupo tan pronto estén disponibles para procesar, y en el momento que las Method Requests cumplan con las restricciones, si es que las hay. Sin embargo, en Android 5.1.1 el Scheduler envía todas las Method Requests al objeto Proxy (clase Handler) y es ahí donde se decide cómo ejecutar las Method Requests. Por lo tanto, la clase Handler debe poseer el grupo de hilos en donde se ejecu-

tan. Este grupo limita la cantidad de hilos que se ejecutan concurrentemente. La cantidad propuesta aquí inicialmente es de 16 hilos. Para una comparación entre el funcionamiento original y la propuesta, considérese nuevamente el esquema presentado en el capítulo 2 sección 2.3, mostrado en la figura 4.1, y el esquema de la propuesta en la figura 4.2.

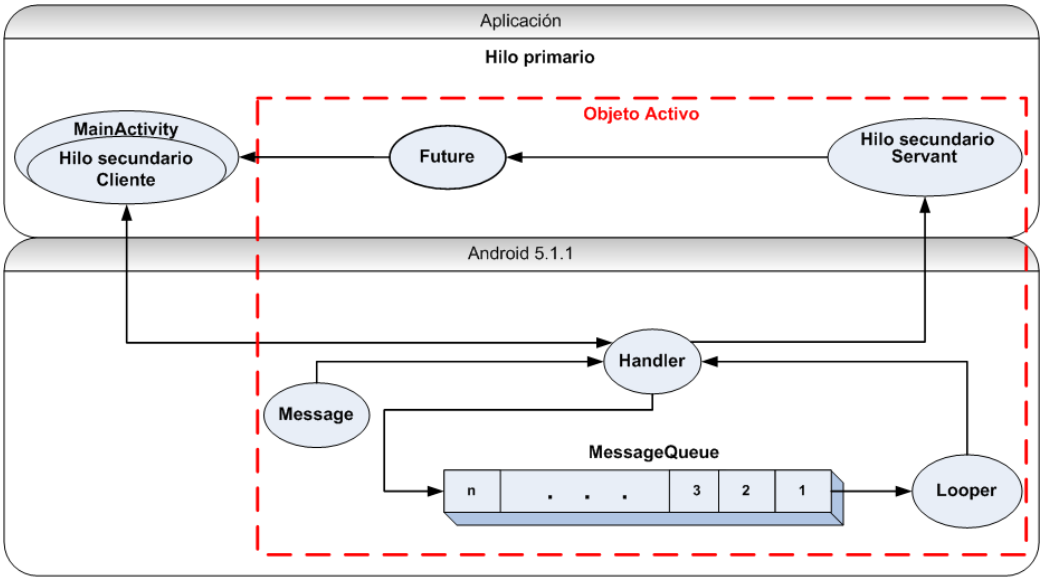

Figura 4.1: Esquema de uso de Objeto Activo en Android 5.1.1.

Para agregar un grupo de hilos a la clase Handler, se propone el uso de la clase Executors que se encuentra definida en el paquete java.util.concurrent de la API de JAVA. Esta clase contiene, entre otras cosas, el método newFixedThreadPool(int nThreads) que devuelve un objeto de tipo ExecutorService, clase definida en el mismo paquete, que contiene finalmente al grupo de hilos (la descripción completa de estas clases se puede consultar en [18]).

En consecuencia, los atributos de la clase Handler deben incluir uno de tipo ExecutorService, y para evitar la creación de una candtidad indefinida de objetos Handler con una cantidad indefinida de grupos de hilos, el atributo es definido como atributo de clase mediante la palabra reservada static. Por lo que existie siempre un solo grupo de hilos compartido por todos los objetos de tipo Handler. Los cambios realizados a la clase Handler quedan de la siguiente forma:

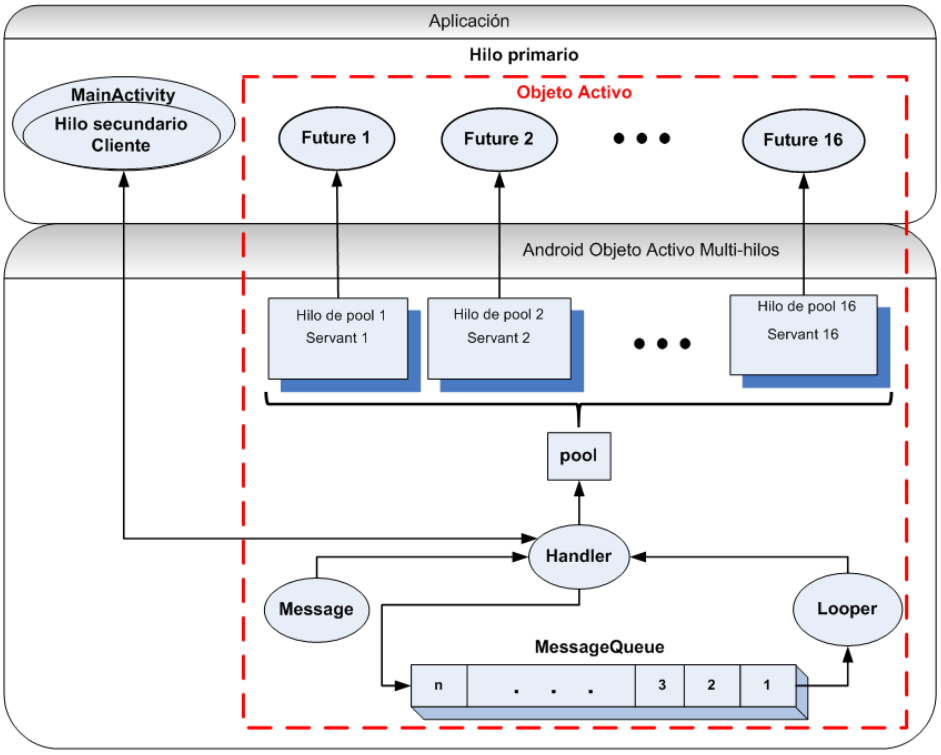

Figura 4.2: Esquema del Objeto Activo Multi-hilos en Android 5.1.1 modificado.

```
//...
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
//...
public final class Handler{
   //...
   public final Message obtainMessage()
   {
      return Message.obtain(this);
   }

   public final Message obtainMessageWithExecutor(Runnable ex)
   {
      return Message.obtainWithExecutor(this, ex);
   }

   public void dispatchMessage(Message msg) {
```

```
            if(msg.executor != null)
            {
               pool.execute(msg.executor);
               return;
            }

            if (msg.callback != null) {
               handleCallback(msg);
            }else {
               if (mCallback != null) {
                  if (mCallback.handleMessage(msg)) {
                     return;
                  }
               }
               handleMessage(msg);
            }
         }
         //...
         //Atributos
         final MessageQueue mQueue;
         final Looper mLooper;
         final Callback mCallback;
         final boolean mAsynchronous;
         IMessenger mMessenger;
         private static final ExecutorService pool =
            Executors.newFixedThreadPool(16);
         //...
      }
```

Para ejecutar las operaciones en el nuevo grupo de hilos de la clase Handler sin afectar el funcionamiento original del sistema operativo, se agrega un nuevo atributo a la clase Message, el cual almacena o referencía a un objeto Servant de tipo Runnable llamado "executor". Este sirve para decidir si se utiliza el nuevo grupo de hilos o no. Además se agrega el método obtainWithExecutor(Handler h, Runnable ex) para obtener un objeto de tipo Message al mismo tiempo que se establece el atributo executor. Este método es muy parecido al original obtain(Handler h). La diferencia es que recibe un argumento Runnable para asignarlo al atributo executor, y es invocado por el método obtainMessageWithExecutor(Runnable ex) de la clase Handler, para obtener una instancia de Message a través de una instancia de Handler,

lo cual asocia al mensaje con esta última.

Además, hay que agregar la línea executor = null; en el método recycleUnchecked() de la clase Message, ya que los objetos de este tipo se reutilizan y dicho método borra el estado de los objetos para la reutilización. De no agregarse, el sistema operativo utiliza estos objetos eventualmente, y al contener un Runnable en el atributo executor, se ejecutan nuevamente, produciendo resultados inesperados. El código modificado de la clase Message es el siguiente:

```
//...
public final class Message implements Parcelable {
   //Atributos
   public int what;
   public int arg1;
   public int arg2;
   public Object obj;
   public Runnable executor;
   //...
   //Metodos
   public static Message obtain(Handler h) {
      Message m = obtain();
      m.target = h;

      return m;
   }

   public static Message obtainWithExecutor(Handler h,
                                            Runnable ex) {
      Message m = obtain();
      m.target = h;
      m.executor = ex;

      return m;
   }
   //...
   void recycleUnchecked() {
   // Mark the message as in use while it remains in the
   // recycled object pool.
   // Clear out all other details.
      flags = FLAG_IN_USE;
```

```
        what = 0;
        arg1 = 0;
        arg2 = 0;
        obj = null;
        executor = null;
        replyTo = null;
        sendingUid = -1;
        when = 0;
        target = null;
        callback = null;
        data = null;

        synchronized (sPoolSync) {
            if (sPoolSize < MAX_POOL_SIZE) {
                next = sPool;
                sPool = this;
                sPoolSize++;
            }
        }
    }
}
```

El método donde se decide cómo ejecutar las operaciones solicitadas a partir de los objetos Message recibidos es el método dispatchMessage(Message msg) de la clase Handler. La modificación consiste en verificar primeramente si el objeto Message contiene algun valor en su atributo executor. De ser así, éste se turna al grupo de hilos para ejecución. De otra forma, se ejecutan las líneas originales:

```
public final class Handler{
    //...
    public void dispatchMessage(Message msg) {
        if(msg.executor != null)
        {
            pool.execute(msg.executor);
            return;
        }

        if (msg.callback != null){
            handleCallback(msg);
        }else {
```

```
            if (mCallback != null) {
               if (mCallback.handleMessage(msg)) {
                  return;
               }
            }
            handleMessage(msg);
         }
      }
      //...
}
```

Para enviar objetos que se ejecuten en los hilos del grupo de la clase Handler, simplemente deben ser objetos de alguna clase que implemente la interfaz Runnable. Por ejemplo, la siguiente:

```
class Servant implements Runnable {

   private int metodo;

   public void setMetodo(int metodo){
      this.metodo = metodo;
   }

   @Override
   public void run(){
      switch(metodo){
         case 1:
            metodo1();
            break;
         case 2:
            metodo2();
            break;
         //...
         case n:
            metodon();
            break;
         default:
      }
   }

   private void metodo1()
```

```
    {
        //...
    }

    private void metodo2()
    {
        //...
    }

    //...

    private void metodon()
    {
        //...
    }
}
```

Después de realizar estos cambios, solo resta mencionar que el método sendMessage(Message msg) de la clase Handler es a través del cual se encolan los mensajes para su posterior ejecución. Un ejemplo de aplicación del nuevo software es el siguiente:

```
public class MainActivity extends ActionBarActivity {
    Servant objServant;
    Handler proxy;
    Mesage mensaje;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        proxy = new Handler();
        objServant = new Servant();
        objServant.setMetodo(1);
        //...
    }

    @Override
    protected void onStart() {
        super.onStart();
```

```
        mensaje = proxy.obtainWithExecutor(objServant);
        proxy.sendMessage(mensaje);
        //...
    }
//...
}
```

4.2. Resumen

En este capítulo, como respuesta al problema planteado en esta tesis, se presenta una alternativa de implementación del patrón Objeto Activo en Android 5.1.1 Lollipop, obteniéndose su variante Multi-hilos. Para lograr eso, se le agrega un atributo a la clase Handler, que contiene un grupo de hilos, en este caso limitado a 16. Para este atributo, se utiliza un objeto de tipo ExecutorService, que se obtiene a través de la clase Executors. Ambas clases se definen en el paquete java.util.concurrent de la API de JAVA. Para no interferir con el funcionamiento original del sistema operativo, el código original se deja completo y se le agrega un atributo a la clase Message, denominado executor, de tipo Runnable, en tiempo de ejecución se utiliza para decidir si se utiliza el grupo de hilos o no, al atributo se le asigna un objeto Servant definido por el desarrollador, que es el que ejecuta el trabajo solicitado por parte del código cliente. Para asignar ese objeto Servant, se agrega un método a la clase Handler y otro a la clase Message. Por último, se modifica el método de la clase Message que recicla sus instancias, para borrar los objetos Servant de los atributos executor. Al final, se presenta un breve ejemplo de uso del nuevo software.

Capítulo 5

Experimentación

En este capítulo se presenta un experimento comparativo simple que contempla el patrón Objeto Activo en su implementación original en el sistema operativo Android 5.1.1 y el mismo patrón modificado con la propuesta de Objeto Activo Multi-hilos, y después de la identificación de la mejor implementación en cuanto a desempeño, se realiza el experimento con una variación en uno de los factores controlables, para obtener una visión del grado de influencia que tiene sobre la salida.

5.1. Descripción del experimento.

La prueba diseñada para experimentación debe comprender la ejecución de algunas tareas que demoren lo suficiente para formar una cola de solicitudes y obtener mediciones significativas que permitan observar las diferencias que se buscan.

La experimentación consiste en enviar un grupo de solicitudes de ordenamiento de los datos contenidos en un archivo de texto plano (extensión .txt) al Objeto Activo. Tales ordenamientos se realizan mediante los conocidos algoritmos de ordenamiento Bubble Sort, Insertion Sort, Merge Sort y Quick Sort, y que están codificados en los correspondientes métodos de una clase Servant. De acuerdo a lo anterior se establece que la entrada del proceso es un archivo de texto que contiene una lista de cadenas de texto, cuyo tamaño es de 500 cadenas (nombres personales). En cuanto a las variables de salida, para cada implementación se tiene el tiempo total de ejecución, y a partir de los resultados de las distintas réplicas se calculan las métricas.

Para obtener esos tiempos se determina que los siguientes son factores controlables:

- Cantidad de peticiones de ordenamiento (Method Requests).
- Tamaño de la lista de cadenas de texto del archivo utilizado como entrada (500 cadenas).
- La cantidad de hilos de la implementación del Objeto Activo Multihilos.

Puesto que aquí se compara el desempeño de una implementación con otra, cada una debe realizar el mismo trabajo, es decir la misma cantidad de ordenamientos mediante cada algoritmo. Como esas cantidades son algunos de los factores de diseño del experimento que se hacen variar, se deben especificar los niveles o cantidades con que se utilizan en el experimento. Los niveles propuestos para cada algoritmo son:

- 1
- 10
- 100
- 1000
- 10000

Acerca del número de experimentos, se realizan 20 por cada configuración del experimento.

La aplicación envía las Method Requests desde un objeto cliente que se ejecuta en un hilo creado desde el hilo principal (figura 5.1). Este objeto utiliza una instancia de Handler para el envío. Las Method Requests son encoladas en la instancia de MessageQueue y la instancia de Looper las desencola enviándolas de nuevo a la correspondiente instancia de Handler. En este caso se envían a un objeto de tipo Servant que implementa la interfaz Callback.

5.2. Herramienta utilizada para el experimento

La aplicación del experimento cuenta con interfaz gráfica de usuario, en la cual se le permite al mismo especificar el archivo en el que se encuentran las cadenas de texto para poder realizar pruebas con distintas listas. El

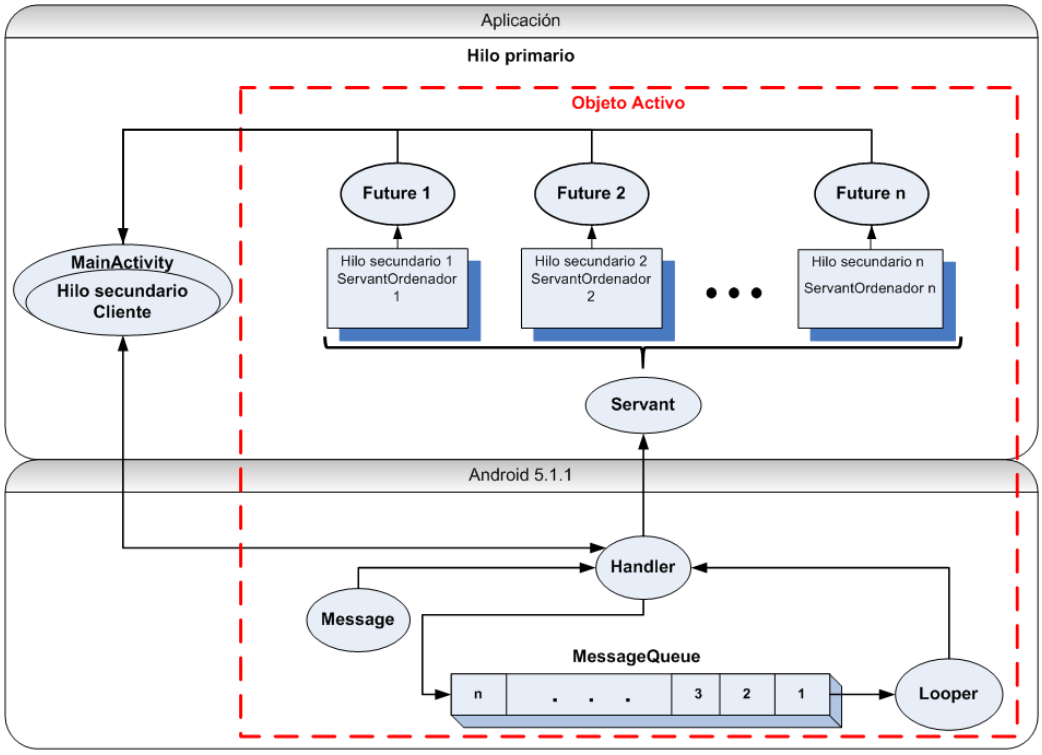

Figura 5.1: Esquema de la aplicación de ordenamientos ejecutada sobre Android 5.1.1.

archivo se carga en memoria mediante un botón con la leyenda "Cargar". También muestra un control NumberPicker para especificar la cantidad de Method Requests que se envían al Objeto Activo. En la parte inferior se muestran barras de progreso (controles ProgressBar) que se visualizan una vez que el usuario indica que se comience la ejecución, lo cual se realiza mediante un botón con la leyenda "Ordenar". Las barras de progreso (una por cada tipo de algoritmo) muestran el avance del procesamiento mientras el objeto atiende las Method Requests (figura 5.3). Una vez que se ha cargado un archivo, es posible reiniciar la aplicación a su estado inicial mediante un botón con la leyenda "Reiniciar". Estos controles se observan en la figura 5.2.

5.3. Método de medición

Dado que los datos de salida están en función de los tiempos relacionados a distintas características del patrón, se utiliza el método currentTimeMillis()

Figura 5.2: Layout principal de la aplicación Objeto Activo Original.

del paquete System para obtener la hora actual del sistema en milisegundos. Este método se invoca antes de ejecutar el código que se desea medir, y después de la ejecución para obtener la diferencia:

```
momentoInicial = System.currentTimeMillis();

    //Código que se desea medir

momentoFinal = System.currentTimeMillis();

tiempoTotal = momentoFinal - momentoInicial;
```

A pesar de que el IDE (Android Studio) cuenta con herramientas para medir tiempos y obtener estadísticas, se prefiere utilizar este método por su simplicidad, ya que las herramientas del IDE consumen tiempo en la obtención de los datos y se realiza durante la ejecución del código medido, lo que

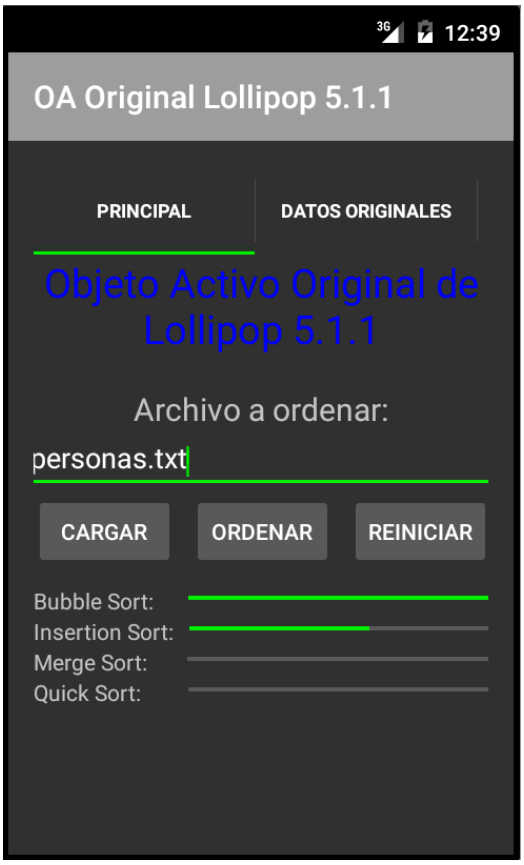

Figura 5.3: Progreso del procesamiento en barras de progreso.

puede afectar los tiempos.

5.4. Descripción de la arquitectura experimental.

Las aplicaciones se ejecutan en un dispositivo emulado, no en un dispositivo real. Este hecho no interfiere con los fines para los que se realiza el presente trabajo, pues siempre que se mida el desempeño del patrón original y el modificado sobre la misma plataforma se están midiendo las diferencias entre patrones. Sin embargo, debido a que los experimentos deben ser reproducibles, es necesario proporcionar las características del ambiente en el que se realizan, y de esta manera, los resultados pueden ser verificados por terceros.

La plataforma sobre la que se emula el dispositivo Android: una PC laptop VAIO SFV-14A15CLS, que posee las siguientes características:

- Procesador Intel CORE i5-3337U, 1.80GHz con Turbo Boost hasta 2.70GHz Intel HM76 Express Chipset, de dos núcleos y dos más virtuales.
- 8 GB de Memoria RAM (velocidad de 1600MT/s).
- Sistema operativo Ubuntu 14.04 LTS de 64 bits.

Sobre esta plataforma se ejecuta el emulador de Android versión 25.1.1, contenido en el paquete de herramientas del Android SDK y las características del dispositivo emulado son:

- Sistema operativo Android 5.1.1 API 22 (Rev 2)
- Procesador x86
- 1GB de memoria RAM

5.5. Resumen

Para comprobar o refutar la hipótesis planteada, es necesario experimentar con los cambios realizados al código de Android. En este capítulo se presenta un experimento comparativo simple que contempla el patrón Objeto Activo en su implementación original en Android 5.1.1 Lollipop y el mismo patrón en su variante Multi-hilos.

Se describe el proceso al que se someten ambas implementaciones, dicho proceso consiste en enviar una cantidad de peticiones al Objeto Activo, para ordenar una lista de cadenas de caracteres mediante cuatro algoritmos de ordenamiento cada vez.

Para el experimento, se determina que existen algunos factores controlables y otros no controlables, de los cuales, los primeros se pueden hacer variar para conocer el grado de influencia que tienen sobre la salida o resultado del proceso. Los factores controlables son: La cantidad de peticiones de ordenamiento, el tamaño de la lista de cadenas de caracteres, y la cantidad de hilos de la implementación Objeto Activo Multi-hilos.

Después de la descripción del experimento, se describe la herramienta que se utiliza, que consiste en una aplicación Android, la cual posee una interfaz de usuario diseñada para permitirle al mismo especificar los datos de entrada (archivo con la lista de cadenas de caracteres), y la cantidad de peticiones, así como un área que muestra los datos ordenados.

Dentro de la aplicación se incluyen algunas líneas de código que permiten medir los tiempos de ejecución. Por último, se describe la plataforma experimental, listando sus características de software y hardware.

Capítulo 6

Medición del desempeño del patrón Objeto Activo y sus componentes en Android 5.1.1

Al realizar el experimento, se obtienen las medidas correspondientes, y se calculan las métricas descritas en la sección 2.4. En la tabla 6.1 se muestra el tiempo promedio de ejecución de las dos implementaciones analizadas, considerando una cantidad de 16 hilos en la variante propuesta.

Peticiones	Tiempo promedio de ejecución de Android 5.1.1	Tiempo promedio de ejecución de Android 5.1.1 Multi-hilos de 16 hilos
1	71.15 ± 18.56 ms	42.40 ± 7.82 ms
10	292.20 ± 84.85 ms	306.55 ± 71.12 ms
100	1,488.60 ± 396.54 ms	674.75 ± 74.95 ms
1,000	12,056.15 ± 3,217.38 ms	4,199.35 ± 107.65 ms
10,000	189,189.40 ± 37,939.33 ms	111,860.90 ± 1,287.31 ms

Tabla 6.1: Tiempo promedio de ejecución del experimento en Android 5.1.1 y Android 5.1.1 Multi-hilos de 16 hilos.

La figura 6.1 presenta la gráfica del comportamiento de ambas implementaciones en el tiempo.

De acuerdo a los datos, se sabe que hay una diferencia de tiempos entre las dos implementaciones, pero además hay que considerar que en algunos casos la diferencia es muy grande. Por ejemplo, cuando se envía una petición

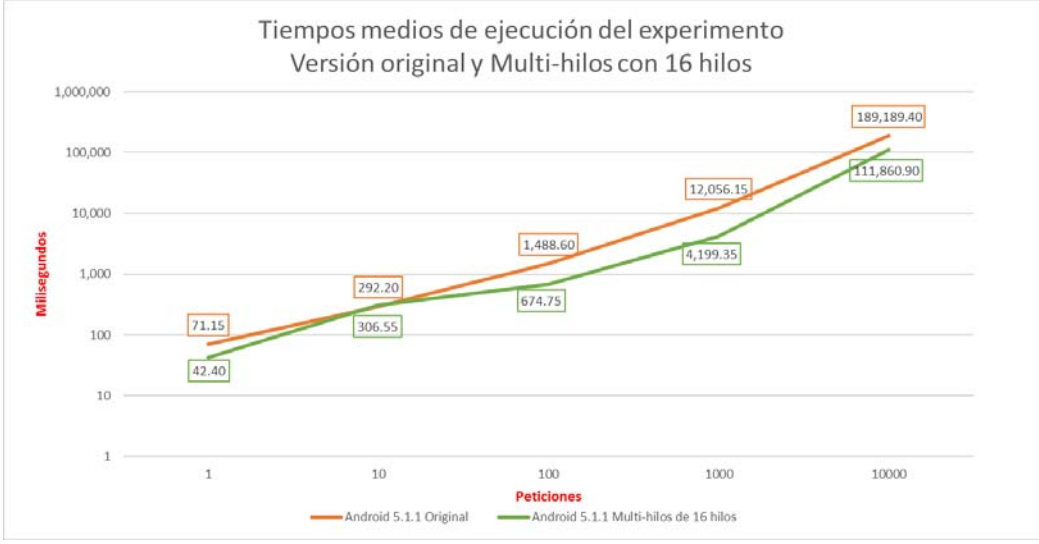

Figura 6.1: Gráfica de los tiempos promedio del experimento en Android 5.1.1 y Android 5.1.1 Multi-hilos de 16 hilos.

o Method Request (por cada tipo de algoritmo), el tiempo de total empleado por la versión multi-hilos equivale al 59.59 % del tiempo total empleado por la versión original, o para la cantidad de 1000 peticiones, que es la cantidad en donde se observa la mayor diferencia, el tiempo de la versión multi-hilos es de tan solo el 34.83 % del tiempo utilizado por el código original.

También se cuenta con la métrica del desempeño promedio, es decir, la cantidad de peticiones procesadas por segundo. Los resultados son mostrados en la tabla 6.2.

Peticiones	Desempeño promedio de Android 5.1.1	Desempeño promedio de Android 5.1.1 Multi-hilos de 16 hilos
1	14.05 $seg^{-}1$	23.58 $seg^{-}1$
10	34.22 $seg^{-}1$	32.62 $seg^{-}1$
100	67.17 $seg^{-}1$	148.20 $seg^{-}1$
1000	82.94 $seg^{-}1$	238.13 $seg^{-}1$
10000	52.85 $seg^{-}1$	89.39 $seg^{-}1$

Tabla 6.2: Desempeño promedio de Android 5.1.1 Multi-hilos con 16 hilos a partir de muestras de tamaño 20 (peticiones procesadas por segundo).

A partir de las métricas obtenidas, es fácil deducir que la implementación multi-hilos ofrece un mejor desempeño. Sin embargo, en busca de optimizar el tiempo de ejecución se hace variar el número de hilos. Con Android 5.1.1 entre 16 y 32 hilos. Los tiempos promedio calculados se muestran en la tabla 6.3. Al igual que para Android 5.1.1 original y su variante Multi-hilos de 16 hilos, la figura 6.2 muestra gráficamente el comportamiento en el tiempo de la variante de 32 hilos, contrastándola en este caso con respecto a la variante de 16 hilos y la original.

Peticiones	Tiempo promedio de ejecución de Android 5.1.1	Tiempo promedio de ejecución de Android 5.1.1 Multi-hilos de 32 hilos
1	71.15 ± 18.56 ms	79.65 ± 9.75 ms
10	292.20 ± 84.85 ms	181.25 ± 31.21 ms
100	1,488.60 ± 396.54 ms	389.00 ± 34.57 ms
1,000	12,056.15 ± 3,217.38 ms	3,618.40 ± 125.57 ms
10,000	189,189.40 ± 37,939.33 ms	101,460.85 ± 5,061.32 ms

Tabla 6.3: Tiempo promedio de ejecución de los tiempos de ejecución del experimento en Android 5.1.1 y Android 5.1.1 Multi-Hilos con 32 hilos.

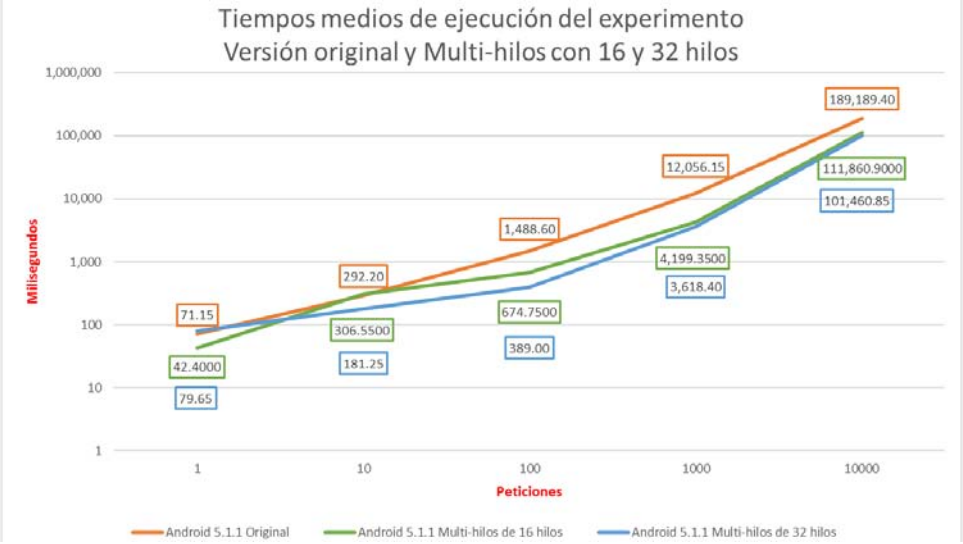

Figura 6.2: Gráfica de los tiempos promedio del experimento en Android 5.1.1 y sus variantes Multi-hilos de 16 y 32 hilos.

La gráfica indica que la versión de 32 hilos comienza con tiempo promedio mayor que la de 16 hilos. Sin embargo, la diferencia es de menos de 40 ms y, antes de la cantidad de 10 peticiones los tiempos se intersectan, y desde este punto en adelante la versión con 32 hilos provee un tiempo promedio menor. El punto en que más se diferencían los tiempos de la implementación original y la de multi-hilos con 32 hilos es cuando se envían 100 peticiones, demorando tan solo el 26.13 % del tiempo utilizado por la versión original.

El desempeño promedio se presenta en la tabla 6.4, recordando que se utiliza un segundo como la unidad de tiempo.

Peticiones	Desempeño promedio de Android 5.1.1	Desempeño promedio de Android 5.1.1 Multi-hilos de 32 hilos
1	14.05 $seg^{-}1$	12.55 $seg^{-}1$
10	34.22 $seg^{-}1$	55.17 $seg^{-}1$
100	67.17 $seg^{-}1$	257.06 $seg^{-}1$
1000	82.94 $seg^{-}1$	276.36 $seg^{-}1$
10000	52.85 $seg^{-}1$	98.56 $seg^{-}1$

Tabla 6.4: Desempeño promedio de Android 5.1.1 Multi-hilos con 32 hilos a partir de muestras de tamaño 20 (peticiones procesadas por segundo).

6.1. Resumen

Al realizar el experimento, se obtienen las medidas necesarias para la obtención de las métricas consideradas. El tiempo promedio de ejecución del Objeto Activo Multi-hilos de 16 hilos, resulta menor que el del original, excepto para una cantidad de 10 peticiones, con la cual se obtiene un tiempo promedio de ejecución ligeramente mayor. Sin embargo, a partir de esta cantidad, la variante propuesta provee de un tiempo promedio de ejecución menor. Con una cantidad de 32 hilos, los resultados son muy similares, la diferencia, es que la propuesta es ligeramente mayor que el código original para una cantidad de peticiones de 1. Ya que el desempeño está relacionado con el tiempo, los resultados obtenidos son los mismos: la variante provee de un mayor desempeño que los componentes originales de Android, a partir de una cantidad mayor a 10 peticiones para una cantidad de 16 hilos, y a partir de una cantidad de peticiones mayor a 1 para una cantidad de 32 hilos.

Capítulo 7

Conclusiones

Para finalizar con este trabajo, aquí se presentan las conclusiones a las que se llega a partir de los resultados obtenidos de la experimentación.

7.1. Reenunciado de la hipótesis

Considerese la hipótesis presentada en la introducción:

> Al adecuar la implementación original del patrón Objeto Activo en Android para obtener una de sus variantes denominada Objeto Activo Multi-hilos (o Thread Pool Active Object en inglés) es posible incrementar su desempeño, pues al contar con múltiples hilos se pueden ejecutar varias tareas concurrentemente, eliminando la necesidad de tener que esperar la finalización de una para comenzar con la siguiente, además de limitar la cantidad de hilos que se pueden crear simultáneamente, evitando la sobrecarga de trabajo del procesador.

7.2. Discusión

A partir de las métricas obtenidas queda claro que el Objeto Activo Multi-hilos realiza un mejor aprovechamiento de los recursos que el originalmente implementado en Android 5.1.1 Lollipop. No importa si se utiliza un grupo de 16 o de 32 hilos. Si en un momento dado, el procesamiento de un conjunto de solicitudes demora lo suficiente para mantener la totalidad de los hilos del grupo ocupados, de modo que las siguientes permanezcan encoladas, se está reduciendo el tamaño de la memoria utilizada por los hilos.

En cuanto al desempeño, las modificaciones realizadas al código superan al original. En ambos casos, con 16 o 32 hilos. El desempeño menor del código modificado cuando se experimenta con 10 peticiones o menos, puede deberse a que un objeto ExecutorService empleado como grupo de hilos (o thread-pool) realiza algunas actividades internas como decidir si crea nuevos hilos (mientras no exceda la cantidad máxima especificada) o utiliza los que ya ha creado para atender las Method Requests entrantes, o almacenar algunos datos estadísticos de las efectivamente atendidas. Eso consume tiempo. En cambio, al crear directamente un nuevo hilo por cada solicitud, se puede reducir el tiempo total ya que en las dos versiones multi-hilos se utiliza una cantidad de hilos mayor a diez.

7.3. Interpretación y análisis de datos

La figura 7.1 muestra las líneas de tendencia de los tiempos de ejecución de las tres implementaciones analizadas.

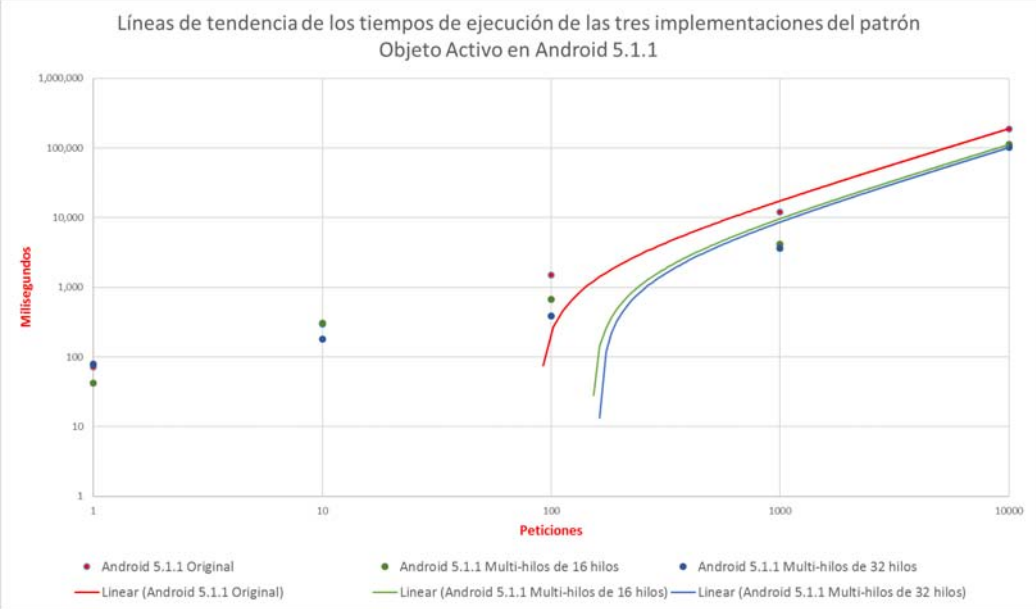

Figura 7.1: Tendencias de los tiempos de ejecución de las tres implementaciones analizadas del Objeto Activo en Android 5.1.1.

Como se menciona anteriormente, los tiempos de ejecución en los distintos hilos son una función no determinista. Si los datos siempre tienen ese

comportamiento, los tiempos de ejecución promedio se listan con el correspondiente intervalo de valores dentro del cual puede encontrarse la media poblacional en un momento dado.

7.4. Re enunciado de las contribuciones

A través de este trabajo se realizan dos contribuciones:

- Un análisis de las deficiencias del tratamiento de concurrencia en la implementación original del sistema operativo Android.
- Una versión modificada del código que implementa el patrón Objeto Activo en Android 5.1.1 Lollipop, que mejore su tiempo de ejecución y desempeño con respecto a la implementación original, incrementando así su responsividad y mejorando la experiencia de usuario.

La primera de ellas, está contenida principalmente en la sección 2.3 y en el capítulo 4, y la segunda, se cubre por completo en el capítulo 4.

7.5. Trabajo futuro

Como trabajo futuro se pueden considerar las siguientes opciones:

- Modificar el código del Objeto Activo de tal manera que las peticiones encoladas en el objeto MessageQueue puedan ser desencoladas por varios objetos Looper simultáneamente. A este patrón formado entre varios objetos que encolan peticiones y varios objetos que desencolan se le conoce como Productor Consumidor con Múltiples Productores y Múltiples Consumidores.

- Para procesar la totalidad de peticiones más rápido, puede resultar de mayor interés para algunos desarrolladores elegir el orden en que se deben procesar, de tal forma que al contar con los resultados de un subconjunto de peticiones éstos puedan ser mostrados al usuario mientras las demás se continúan procesando, incrementando así la responsividad. Para lograr esto, se agregarán atributos o métodos "guardias" a los objetos utilizados como peticiones para convertir al objeto MessageQueue en una cola de prioridades, dichos guardas serán evaluados por el planificador del patrón, es decir, la clase Looper.

Referencias

[1] Gamma Erich, et al (1995). Design Patterns, Object-Oriented Software, Addison Wesley. (395 p.).

[2] Bass L., et al (2003). Software Architecture in Practice, Addison Wesley (560 p.)

[3] Buschmann F., et al (1996). Pattern-Oriented Software Architecture, A system of patterns, volume 1, John Wiley and Sons, Ltd. (467 p.).

[4] Schmidt D., et al (2000). Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, volume 2, John Wiley and Sons, Ltd. (633 p.).

[5] Jani N.N., et al (2009). Mobile Computing (Technologies and Applications), S. Chand and Company Ltd. (231 p.).

[6] Gironés J. T., (2012). El gran libro de Android, Alfaomega (404 p.).

[7] Zheng P., et al (2006). Smartphone and Next Generation Mobile Computing, Elsevier Inc. (552 p.).

[8] Alexander Christopher, et al (1977). A Pattern Language, Oxford University Press (1171 p.).

[9] Alexander Christopher, (1979). The Timeless Way of Building, Oxford University Press (552 p.).

[10] Schmidt D., (1994). Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching, Proceedings of PLoP '94.

[11] Object Management Group: CORBAservices: Common Object Services Specification OMG Document Number 95-3-31, (1995)

[12] R. Blaha M., et al (1990). Object-Oriented Modeling and Design, Prentice-Hall (512 p.).

[13] García Valls M., Crespo A., Vila J., (2013), Resource Management for Mobile Operating Systems based on the Active Object Model, Universidad Carlos III de Madrid, Universidad Politécnica de Valencia, España.

[14] Ortega Arjona J. L., Graham R., (2001), Pattern-Based Simulation: Simulating the Actor Model Using the Active Object Behavioural Pattern, Department of Computer Science, University College London, England.

[15] https://developer.android.com/reference/packages.html

[16] C. Montgomery D., (2004). Diseño y análisis de experimentos, Limusa Wiley (686 p.).

[17] S. Pressman, R., (2006). Ingeniería del Software Un Enfoque Práctico. McGraw-Hill (957p.).

[18] http://docs.oracle.com/javase/8/docs/api/index.html

[19] Ortega Arjona J. L. Una introducción a los patrones de software, Departamento de Matemáticas, Facultad de Ciencias, UNAM.

[20] IEEE Standard Glossary of Software Engineering Terminology, IEEE, 1993.

[21] Agha, G., Frolund, S., Kim, W.Y., Panwar, R., Patterson, A., and Sturman, D. (1993). Abstraction and Modularity Mechanisms for Concurrent Computing. In Gul Agha, Peter Wegner and Akinori Yonezawa, (eds.) Research Directions in Concurrent Object-Oriented Programming. The MIT press.

[22] Agha, G., Mason, LA., Smith, S.F., and Talcott, C.L. (1993). A foundation for Actor Computation., Journal of Functional Programming, Vol. 1. Cambridge University Press.

[23] Stone, H.S., et al (1975). Introduction to Computer Architecture. Science Research Associates, Inc.