UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Enfoques de desarrollo de Aplicaciones Móviles: un análisis de parámetros para la toma de decisiones.

Autores: Juan Alfonso Cuitiño

Directores: Lisandro Delía, Pablo Thomas

Carrera: Licenciatura en Informática

Resumen

La masificación de los dispositivos móviles con gran capacidad de cómputo ha traído gran relevancia al desarrollo de aplicaciones móviles. Este tipo de desarrollo tiene características propias que no estaban presentes en el desarrollo tradicional de software. Deben tenerse en cuenta requisitos como los tiempos del mercado, las limitaciones de los dispositivos móviles, la diversidad de plataformas, entre muchos otros. Para maximizar la presencia en el mercado, puede optarse por desarrollar aplicaciones específicas para cada plataforma, con varios desarrollos en paralelo usando herramientas y lenguajes propios de cada una, lo que se denomina enfoque nativo. Otra opción es realizar un desarrollo único, usando herramientas que permitan llevar la aplicación a más de una plataforma, lo cual se denomina enfoque multiplataforma. La presente tesina consiste en la investigación de los diferentes enfoques de desarrollo de aplicaciones móviles, y el desarrollo de experimentos para conformar un marco comparativo entre los enfoques. Se implementaron aplicaciones en todos los enfoques estudiados, y compararon según parámetros que pueden influir en el éxito de una aplicación, tales como el uso de energía de la batería o el espacio de almacenamiento ocupado.

Palabras Claves

Dispositivos móviles. Aplicaciones móviles nativas. Aplicaciones móviles multiplataforma. Consumo de batería. Uso de espacio de almacenamiento.

Trabajos Realizados

Se estudiaron los diferentes enfoques de desarrollo de aplicaciones móviles, y los parámetros que influyen en el éxito de una aplicación. Se diseñaron experimentos en donde se implementaron aplicaciones con idéntica funcionalidad en cada uno de los enfoques, con el objetivo de comparar los resultados, teniendo en cuenta los parámetros elegidos.

Conclusiones

Mediante la investigación, el desarrollo de los experimentos y el análisis de los resultados, se logró comparar los diferentes enfoques de desarrollo de aplicaciones móviles. Se usaron como parámetros de comparación el consumo de batería y el uso del espacio de almacenamiento de los dispositivos. Este análisis se presenta como un marco comparativo para elegir un enfoque, al momento de iniciar el desarrollo de una aplicación móvil.

Trabajos Futuros

Se propone como trabajo futuro el estudio de nuevos frameworks multiplataforma presentes en el mercado. Además, se plantea el análisis de otros parámetros de comparación, tal como la experiencia de usuario final, el acceso a disco, y el tiempo de carga de las vistas.

Fecha de la presentación: Abril de 2018

Universidad Nacional de La Plata

Facultad de informática

Enfoques de desarrollo de Aplicaciones Móviles: un análisis de parámetros para la toma de decisiones

Tesina de Licenciatura en Informática

Alumno:

Juan Alfonso Cuitiño

Directores:

Esp. Lisandro Delía

Mg. Pablo Thomas

Índice de contenidos

Índice de figuras

Índice de tablas

Capítulo 1. Introducción

1.1 Motivación

La masificación de los dispositivos móviles con buenas capacidades de cómputo pone en el foco de atención el desarrollo de aplicaciones móviles, exigiendo funcionalidades cada vez más avanzadas. Por otro lado, los usuarios esperan que las aplicaciones sean eficientes en el uso de aquellos recursos que son limitados en sus dispositivos, como la batería, el espacio de almacenamiento y los datos transmitidos por la red celular, entre otros. Muchos estudios consideran el ahorro de energía de la batería como uno de los requerimientos no funcionales más importantes a tener en cuenta en el desarrollo de aplicaciones móviles [1].

Para que una aplicación tenga éxito, otro de los requerimientos fundamentales es que pueda ejecutarse en la mayor cantidad de plataformas posible. Actualmente, la mayor concentración del mercado móvil se divide entre Android (72%) y iOS (19.5%) [2].

En la actualidad existen diversas herramientas para el desarrollo de aplicaciones móviles. Al momento de iniciar un proyecto de aplicación móvil, se debe elegir una de ellas, teniendo en cuenta una diversidad de factores como el uso de un lenguaje de programación conocido por los programadores, disponibilidad de herramientas y entornos de desarrollo, el soporte para diversas plataformas, la performance, experiencia de uso, satisfacción del usuario, entre otras características.

Por un lado se encuentra el enfoque de desarrollo Nativo, donde la aplicación es desarrollada usando las herramientas oficiales de cada plataforma, como Java y Android SDK para el sistema operativo Android, y Objective C y Swift para el sistema operativo iOS. De esta forma, se pueden aprovechar las funcionalidades nativas de cada plataforma en su totalidad, pero no se puede compartir código entre las plataformas, sino que se debe encarar el desarrollo de una aplicación desde cero para cada una.

Asimismo, existen herramientas que permiten desarrollar la aplicación una única vez, obteniendo ejecutables para varias plataformas. Estas se denominan aplicaciones multiplataforma. Algunos ejemplos son Apache Cordova [3], Ionic [4], Appcelerator Titanium [5], Nativescript [6], entre otras. Esto se logra con diferentes enfoques, como por ejemplo el enfoque Híbrido, donde se programa usando tecnologías web como HTML, JavaScript y CSS, que luego se ejecutan dentro de un contenedor web nativo; o el enfoque Interpretado, donde se generan automáticamente componentes nativos de la interfaz, mientras que la mayor parte de la lógica de la aplicación está implementada en otro lenguaje común a todas las plataformas, como es el caso de JavaScript en Appcelerator Titanium.

Al momento de iniciar el desarrollo de una nueva aplicación móvil, el ingeniero de software debe tomar la decisión de cuál enfoque de desarrollo utilizar, teniendo en cuenta todos los factores mencionados.

1.2 Objetivos

Existen diversos enfoques de desarrollo de aplicaciones móviles. El objetivo principal de este trabajo consiste en el estudio de estos enfoques.

Se propone elaborar un estudio comparativo, con el fin de analizar parámetros que influyan en el potencial éxito de una aplicación móvil. De esta forma, se busca lograr un marco comparativo que facilite la selección del enfoque de desarrollo de aplicaciones móviles.

Se tomará en cuenta la taxonomía de enfoques de desarrollo descripta en [7]. En base a éstos, se desarrollarán aplicaciones con idéntica funcionalidad siguiendo cada uno de los enfoques, con el objetivo de comparar las aplicaciones resultantes según una serie de parámetros a definir.

1.3 Estructura de la tesina

Esta tesina se organiza de la siguiente manera:

- En el capítulo 1 se describe el objetivo del trabajo y se resume la motivación.
- En el capítulo 2 se profundiza el marco teórico del trabajo, la motivación, el contexto de la tecnología y los diferentes enfoques de desarrollo de aplicaciones móviles.
- En el capítulo 3 se presentan los experimentos diseñados y sus objetivos.
- El capítulo 4 se detallan los resultados de los experimentos realizados.
- En el capítulo 5 se presentan las conclusiones obtenidas y se proponen trabajos futuros.
- Finalmente, se presenta la bibliografía utilizada.

Capítulo 2. Marco teórico

2.1 Tecnología móvil

Una aplicación móvil es un programa informático diseñado para ejecutarse en dispositivos con movilidad física, tales como teléfonos inteligentes y tabletas. Permite al usuario realizar tareas específicas de diferentes tipos: profesionales, recreativas, educativas, de acceso a servicios, etc., con la libertad de ejecutarse en un dispositivo portable [8].

Hasta principios de la década del 2000, los teléfonos móviles solo contaban con la funcionalidad de realizar llamadas. Esto fue cambiando con la aparición de dispositivos tipo PDA (*Personal Digital Assistant*) y los primeros teléfonos inteligentes con sistemas operativos como Symbian y Blackberry OS, que popularizaron el acceso a internet móvil, y el concepto de aplicaciones móviles instaladas por el usuario, a diferencia de los dispositivos con aplicaciones preinstaladas por el fabricante, con las cuales no había posibilidad de modificarlas o agregar nuevas.

Más adelante, fueron apareciendo otras opciones tales como los dispositivos con sistema operativo iOS fabricados por Apple, y una gran variedad de dispositivos con Android y Windows Phone, los cuales son fabricados y comercializados por una gran cantidad de fabricantes.

Este crecimiento amplió y mejoró las posibilidades de uso de las aplicaciones móviles y sus campos de aplicación.

En los últimos años, el uso de dispositivos móviles fue creciendo aceleradamente, tanto es así que durante 2013, como se ve en la figura 1, la cantidad de usuarios a nivel global de dispositivos móviles superó a la de usuarios de computadoras de escritorio [9].

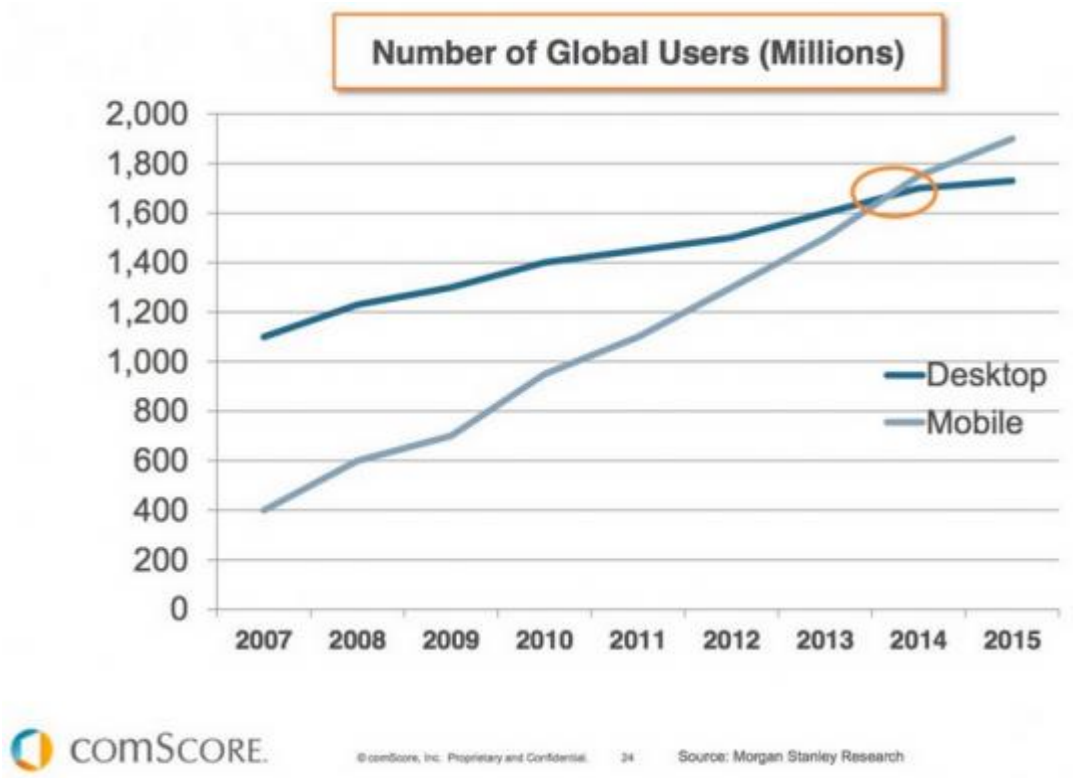


Figura 1. Estadística de cantidad de usuarios de internet a nivel global

En Argentina, a mayo de 2017, el acceso a internet sigue la misma tendencia, con un 47,32% de las personas que usan internet accediendo desde dispositivos móviles, y un 52,38% accediendo desde computadoras de escritorio (ver figura 2).

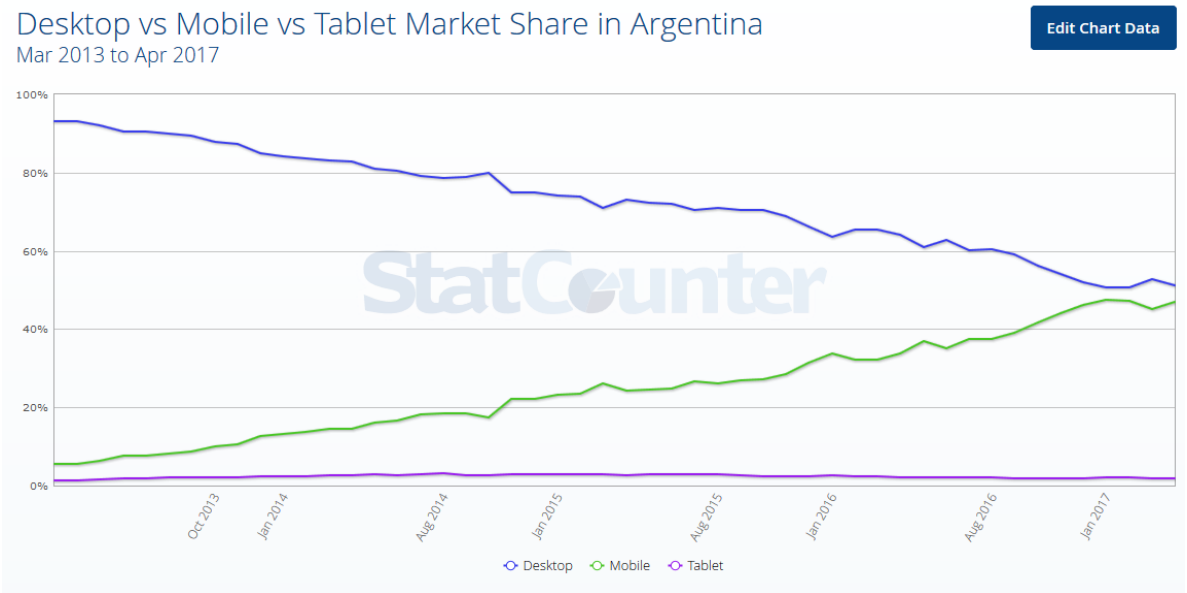


Figura 2. Porcentaje de dispositivos usados para acceder a internet

Esta tendencia también se ve en el creciente uso de aplicaciones, la cantidad de horas que los usuarios pasan en sus dispositivos móviles se incrementa a un ritmo sostenido año a año [10].

En los últimos años, los dispositivos móviles han evolucionado considerablemente en muchos aspectos, especialmente la capacidad computacional y las posibilidades de su uso. Existen múltiples fabricantes que compiten por ofrecer las tecnologías más avanzadas y llegar a la mayor cantidad de usuarios posibles.

Hoy en día, los teléfonos inteligentes incorporan lo que se conoce como SoC (del inglés *System on a Chip*, Sistema en un Chip), una tecnología que consiste en incorporar una gran cantidad de los componentes que conforman un sistema de cómputo, en un único sistema integrado o chip. En el caso de los dispositivos móviles, se los optimiza para lograr la mayor eficiencia energética posible intentando mantener un alto poder de cómputo y funcionalidades.

Por ejemplo, el SoC “Snapdragon 835” fabricado por la empresa Qualcomm [11], incluye un procesador de 8 núcleos con arquitectura de 64 bits y hasta 2.45 GHz de velocidad, una unidad de procesamiento gráfico (GPU), un módem para comunicaciones por tecnología celular, WiFi, NFC, Bluetooth, GPS, unidades de procesamiento de imágenes, audio y video 4K, soporte para interfaces de entrada USB 3.1, control de carga de la batería, entre muchas otras funcionalidades.

Estos sistemas son adaptados para cada modelo de dispositivo móvil. Actualmente, se pueden encontrar dispositivos en el mercado que apuntan a ser económicos con el fin de llegar a la mayor cantidad de usuarios posible, con 1GB de memoria RAM y pantallas de aproximadamente 4 pulgadas, y otros orientados a tener la máxima performance, que cuentan con procesadores de 8 núcleos y hasta 8GB de memoria RAM, junto con otros componentes para proveer funcionalidades avanzadas, tales como sensores de presión sobre el dispositivo, lectores de huellas dactilares, reconocimiento facial, entre muchas otras.

El espacio de almacenamiento varía ampliamente entre los diferentes modelos de smartphones. Si bien existen algunos con hasta 256 GB de memoria interna, la gran mayoría de los dispositivos económicos ronda los 8 ó 16 GB. El problema con el que se encuentran los usuarios de estos últimos es que el sistema operativo y las aplicaciones pre-instaladas de fábrica ocupan una gran parte del espacio. Esto limita las posibilidades de su uso [12] y hace que los usuarios sean reacios a instalar nuevas aplicaciones, o incluso dejen de utilizar aquellas que ocupen mucho espacio.

Un análisis de estudios de experiencia de usuario realizados por Google mostró que el tamaño de las aplicaciones publicadas en la tienda de aplicaciones Play Store se ha quintuplicado entre 2012 y 2017 (ver figura 3) [13].

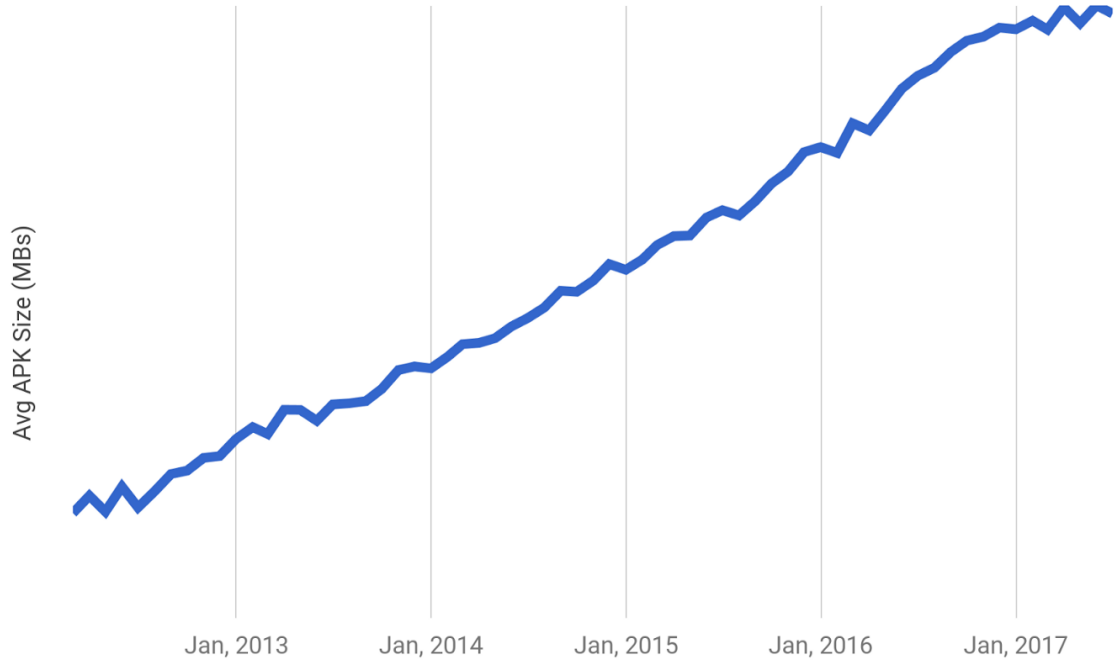


Figura 3. Tamaño promedio de las aplicaciones en MegaBytes, en Google Play Store

Esto se debe a que a medida que el mercado madura, los usuarios requieren nuevas funcionalidades, recursos multimediales de mejor calidad, etc. Sin embargo, se encontró que los usuarios le dan importancia al tamaño de la aplicación antes de descargarla. Se observó que a medida que las aplicaciones crecen, por cada 6 megabytes se reduce en 1% la cantidad de usuarios que efectivamente instalan la aplicación después de leer su información. Además, se observó que una vez que el usuario decide iniciar la descarga, aquellas aplicaciones con 100 megabytes de tamaño tienen un 30% menos de descargas completadas, comparado con aplicaciones de 10 megabytes. Esto puede ocurrir por varias razones, siendo la más obvia los problemas de conectividad que ocurren en las redes móviles, y por otro lado que los usuarios son reacios a utilizar transferencia de datos a través de la red móvil, tanto por la velocidad inferior a la de una conexión por cable, como por el costo que puede significarle. El análisis muestra que la ubicación geográfica también influye. Aquellos usuarios que se encuentran en países en vías de desarrollo descargaron en promedio aplicaciones con la cuarta parte del tamaño, comparado con usuarios que se encuentran en países desarrollados (ver Figura 4). Esto se debe a un menor desarrollo de las infraestructuras de conectividad, menor nivel de acceso permanente a redes WiFi y el costo más alto del uso de datos móviles. Teniendo en cuenta estos datos, se compararon las reacciones de los usuarios con respecto a un decremento de 10 megabytes en el tamaño de la aplicación. Entre los usuarios en India y Brasil se vio un incremento del 2,5% en la cantidad de instalaciones exitosas, mientras que en Alemania y Estados Unidos, el aumento fue de 1,5% [13].

Por todo esto, resulta fundamental que los desarrolladores de aplicaciones tengan en cuenta el uso del espacio de almacenamiento para llegar a mayores audiencias.

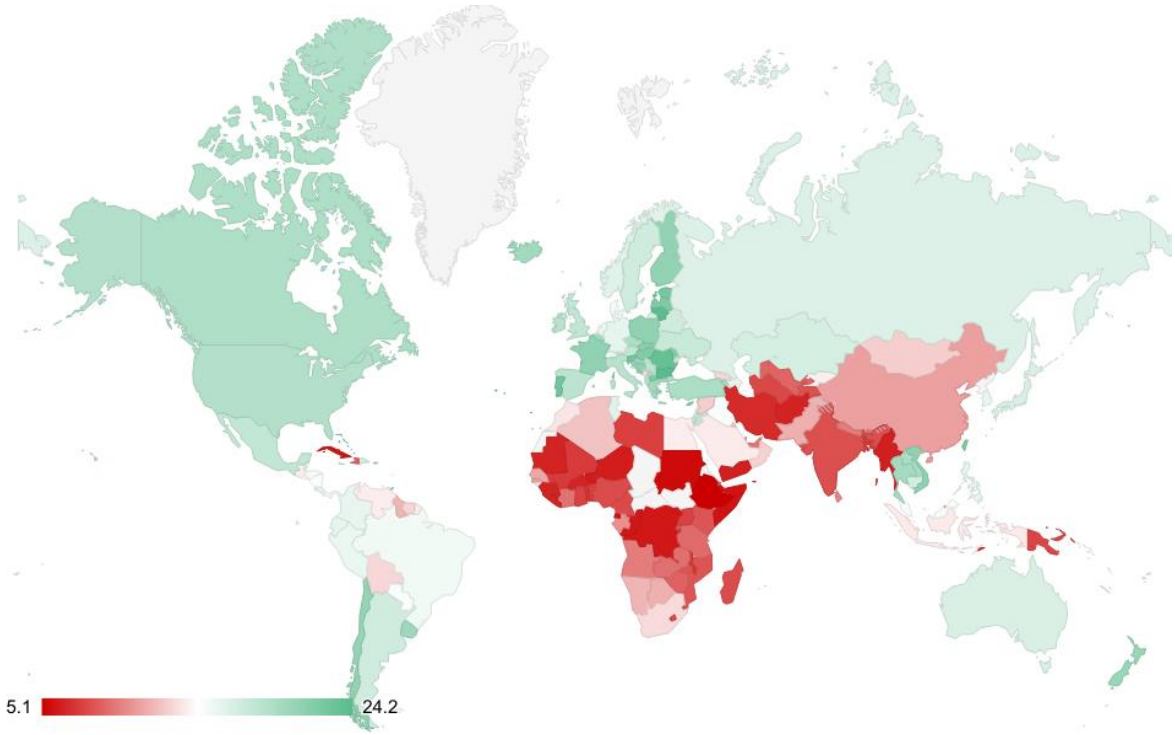


Figura 4. Tamaño promedio de APK descargado. Verde implica mayor promedio, y rojo menor promedio

Los dispositivos móviles utilizan almacenamiento basado en memoria flash, a diferencia de las computadoras personales que usan discos rígidos magnéticos de alta capacidad.

Al estar basada en semiconductores NAND, las memorias flash proveen gran velocidad de lectura, pero pueden sufrir de baja performance al realizar lectoescrituras aleatorias [14]. Esto implica que si una aplicación tiene mucha dependencia del almacenamiento, se impactará negativamente la experiencia de usuario, ya que podrán notarse demoras para realizar las tareas.

Por el lado de las baterías, se ve una evolución de las tecnologías menos pronunciada que en el resto de los componentes. Los fabricantes van aumentando lentamente la capacidad de las baterías año a año, pero sin diferencias revolucionarias [15].

Hoy en día, la industria de las baterías para dispositivos móviles se centra en las baterías de iones de litio, que a diferencia de otras tecnologías como las de níquel-cadmio, presenta mejor capacidad energética, poco efecto memoria y tamaño reducido.

Sin embargo, la mayoría de los teléfonos inteligentes tiene una duración esperada de la carga de la batería de entre uno y dos días de uso. Esto se

debe a que todos los demás componentes dependen del suministro de energía de la batería, especialmente el procesador y la pantalla, creando un compromiso entre el poder de procesamiento y la duración de la batería. Es decir, si una aplicación hace demasiado procesamiento de datos, estará limitando el tiempo que el usuario puede usar el dispositivo.

La tendencia actual es integrar cada vez más aplicaciones en un único dispositivo móvil de uso general, lo que a menudo resulta en un consumo de energía mayor y, por consiguiente, una menor duración de la carga de las baterías. Este mayor consumo plantea problemas para la evolución de la computación móvil, ya que los desarrolladores no pueden utilizar todo el potencial de la tecnología actual.

Si un usuario detecta que una aplicación usa mucha energía (ver Figura 5), puede tomar la decisión de eliminarla o reemplazarla por otra más eficiente, ya que un alto consumo puede significarle la imposibilidad de llegar al final del día sin tener que recargar el dispositivo.

Un estudio realizado sobre 170000 revisiones de usuarios recopiladas en la tienda de aplicaciones de Google Play, reveló que los usuarios son más propensos a desinstalar una aplicación si muestra un comportamiento ineficiente en cuanto a energía en comparación con otros tipos de comportamientos ineficientes [16].

Además, la vida útil de las baterías de litio está determinada por la cantidad de ciclos de carga y descarga, esto es, la cantidad de veces que el dispositivo se carga y descarga completamente [17]. De esto se deduce que mientras más intensivamente se utilice el dispositivo, más rápido se deteriorará la batería.

El compromiso por alcanzar una gestión eficiente de la energía está por encima de la necesidad de satisfacer las demandas de los usuarios de las aplicaciones móviles.

El mayor consumo de energía está en contra de la tendencia actual de la informática verde (*green computing*), que intenta conseguir más ahorro de energía y sistemas computacionales amigables con el medio ambiente. Prolongar la duración de la carga de las baterías y su vida útil tiene un efecto positivo sobre el ambiente reduciendo la contaminación derivada de los procesos de generación de energía y fabricación o desecho de los materiales involucrados en las baterías.

Varios estudios han abordado esta cuestión desde el punto de vista del hardware y el software intentando determinar cómo reducir el impacto ambiental relacionado con la utilización cada vez más masiva de los dispositivos móviles.

En [18] se discute la creación de técnicas para optimizar el consumo de energía de los dispositivos móviles y también se propone un programa de certificación para validar si un dispositivo puede ser considerado "verde". Según los autores del artículo, la creación del programa de certificación móvil

verde (*green mobile certification*) provocará que los desarrolladores consideren la característica de ahorro de energía como un requisito no funcional de sus sistemas, sumándose así a las tendencias actuales de generar productos ecológicos. Para ello, es necesario conocer cómo diseñar software *eco-friendly*, para crear aplicaciones que satisfagan al usuario mientras hacen un uso inteligente de los recursos energéticos.

Desde el lado del hardware, el fabricante debe tener en cuenta este compromiso generado por las baterías, para la elección de las demás características. Por ejemplo, la tecnología utilizada en la pantalla impactará directamente en la duración de la batería [19]. Algunos fabricantes incluyen pantallas de resolución 4K, pero permiten que el usuario cambie entre resolución 4K y 2K con el objetivo de ahorrar energía.

La introducción de la tecnología big.LITTLE del fabricante de SoC y procesadores ARM [20] para abordar la necesidad de un mejor rendimiento y eficiencia energética en los dispositivos móviles constituye otro ejemplo del esfuerzo que hacen en este sentido los fabricantes de hardware. Esta consiste en una arquitectura de cómputo heterogénea compuesta por núcleos de bajo rendimiento y bajo impacto en la batería, y núcleos más potentes y con más consumo. El objetivo es crear un procesador multicore que pueda responder dinámicamente a la demanda del sistema, con el mínimo impacto en el consumo de energía. ARM asegura que para ciertas actividades, logra ahorrar hasta 75% de energía.

Otro ejemplo son los dispositivos móviles equipados con un *assisted*-GPS (A-GPS), que mejora la precisión, rendimiento y eficiencia energética en relación al GPS convencional [21].

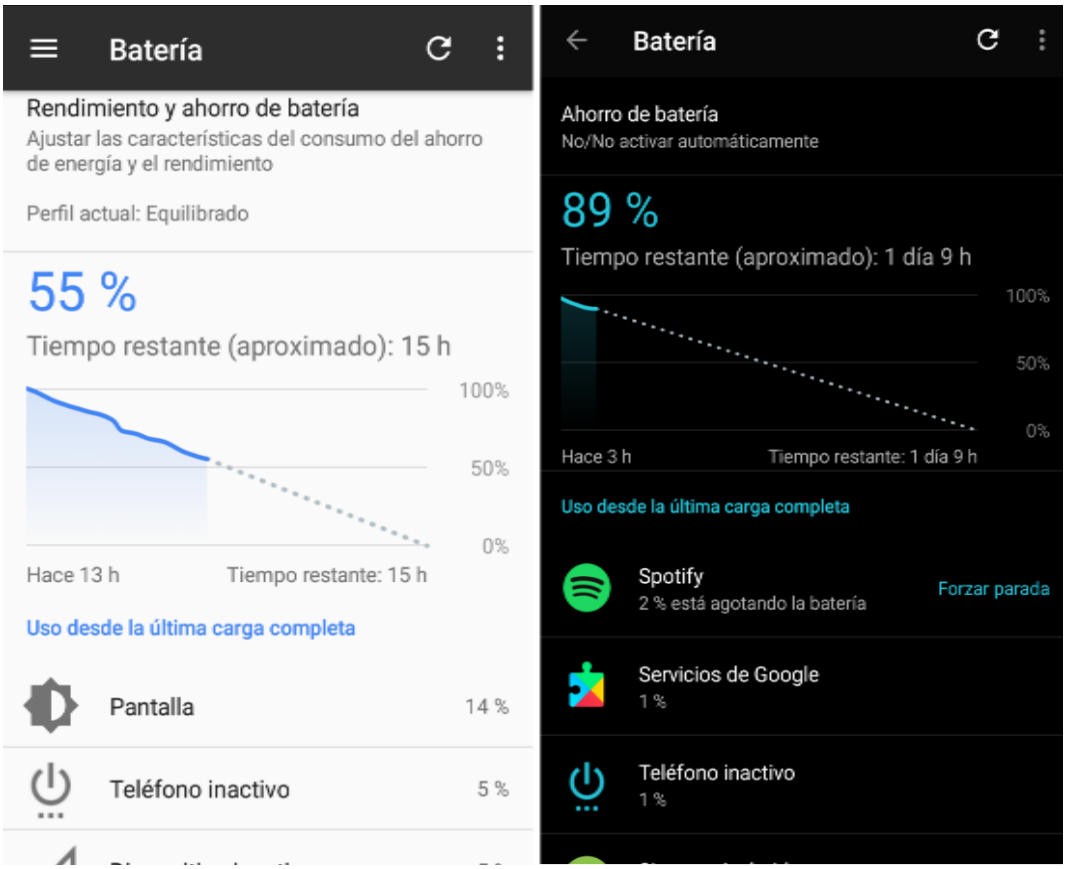


Figura 5. El inspector de batería de Android, que permite al usuario ver cuánto porcentaje de energía consumió cada aplicación o componente, y detener a las que más consuman.

El creciente uso de las aplicaciones ha naturalizado el hecho de se dejen de lado otros medios, y se usen los teléfonos inteligentes para una gran cantidad de las actividades diarias de las personas. Por ejemplo, comunicación [22], compartir fotografías, leer correo electrónico, escuchar música, usar redes sociales, realizar compras online, entre muchas otras. Incluso se han creado nuevas categorías como aplicaciones para acompañar el ejercicio físico [23], cuidado de pacientes [24], asistencia al manejo, aprendizaje de idiomas, etc.

Todo esto lleva a que los usuarios tengan altas expectativas de sus dispositivos y aplicaciones, requiriendo funcionalidades cada vez más avanzadas. Así, se pone el foco aún más sobre la importancia de que los desarrolladores elijan las herramientas correctas para encarar el desarrollo, y lograr un balance en el compromiso entre rendimiento, funcionalidades, consumo de energía, espacio de almacenamiento y demás recursos limitados.

En un mercado de aplicaciones móviles tan variado como el actual, resulta fundamental para los desarrolladores tener el mayor alcance posible. Esto implica poder llegar a todas las plataformas móviles.

El mercado de plataformas móviles se encuentra abarcado en su mayor parte por los sistemas operativos Android y iOS. A nivel global [2], en julio de 2017, Android tenía un 70,2% del mercado, e iOS un 22,56%. En Argentina [25], el

porcentaje pasaba a ser de 87,38% para Android y 8,31% para iOS, y en tercer lugar 3,22% para Windows Phone.

El medio más popular para distribuir aplicaciones es a través de tiendas de aplicaciones. Una tienda de aplicaciones es un medio de distribución digital de software, que consiste en una aplicación que permite visualizar información sobre otras aplicaciones e instalarlas, además de proveer otras funcionalidades como búsquedas, filtrados, reseñas, calificaciones, etc.

Actualmente, las tiendas de aplicaciones más grandes [26] son el Apple App Store, con cerca de dos millones de aplicaciones, y Google Play Store, con tres millones. [27] En tercer lugar, se encuentra Windows Store de Microsoft (antes Windows Phone Store), que concentra aplicaciones que funcionan tanto en Windows Phone como en Windows para sistemas de escritorio, y tiene aproximadamente un millón de aplicaciones. Otra tienda de aplicaciones que está ganando popularidad en los últimos años es Amazon Appstore, que a diferencia de las anteriores que son propias de una empresa para su plataforma, Amazon se enfoca en distribuir aplicaciones para Android.

2.2 Enfoques de desarrollo de aplicaciones móviles

Para poder ejecutarse en cada una de las plataformas, una aplicación debe ser compilada con el conjunto de herramientas que provee cada uno de los respectivos fabricantes, comúnmente llamado SDK (*Software Development Kit*).

El SDK está diseñado para ser utilizado con un lenguaje de programación que el fabricante defina, como es el caso de Java o Kotlin para Android, u Objective C o Swift para iOS. Este conjunto de librerías, lenguajes de programación y demás herramientas son llamadas herramientas nativas.

Esto tiene la desventaja de que al usarse un lenguaje de programación diferente en cada plataforma, se deberá contar con programadores que sepan cada uno de esos lenguajes, ocasionando más costos económicos y de tiempos.

Otro enfoque consiste en desarrollar una única vez usando herramientas multiplataforma.

En éstas, el objetivo es escribir el código una única vez, y poder ejecutarlo en los diferentes sistemas operativos, disminuyendo así los costos mencionados anteriormente. Las herramientas para desarrollar aplicaciones multiplataforma pueden ser clasificadas en web, híbridas, de compilación cruzada e interpretadas.

2.2.1 Enfoque Nativo

Para alcanzar la mayor cantidad posible de usuarios, se busca que una aplicación pueda ser ejecutada en la mayoría de las plataformas móviles disponibles.

El primer enfoque es desarrollar aplicaciones usando las herramientas nativas que provee cada plataforma, como es el caso de Android SDK y Java para Android, Swift para iOS, o Windows SDK con C# para Windows.

El desarrollo se lleva a cabo utilizando los IDE (*Integrated Development Environment*) provistos por los desarrolladores de los sistemas operativos. Gracias a esto, se cuenta con múltiples herramientas para facilitar el desarrollo y puesta en producción de las aplicaciones.

El desarrollo nativo permite un acceso óptimo a todas las funcionalidades de los dispositivos, tales como GPS, cámara, acelerómetro, y demás sensores, y componentes del sistema como hilos de ejecución para procesamiento paralelo, o interfaces de entrada tales como paneles táctiles.

La principal desventaja de este enfoque es que no se puede reutilizar el código. Por ejemplo, si se desarrolló la aplicación usando Swift para iOS, al momento de llevar la misma funcionalidad a Android o Windows Phone, se deberá

comenzar el desarrollo nuevamente, usando las herramientas correspondientes a cada plataforma.

Esto deriva en un mayor costo en recursos económicos y de tiempo, ya que se debe contar con desarrolladores que tengan conocimientos en todas las plataformas involucradas, o distintos desarrolladores para cada plataforma, y mantener varios desarrollos en paralelo.

2.2.1.1 Desarrollo nativo en Android

El sistema operativo Android es el más popular en la actualidad, pudiendo ejecutarse en teléfonos inteligentes, tablets, relojes inteligentes, automóviles y televisores, entre otros dispositivos. Está basado en el kernel 3.0 de Linux, y es activamente desarrollado por la empresa Google.

El desarrollo de aplicaciones para Android se hace utilizando el lenguaje de programación Java, junto con el conjunto de herramientas provisto por el Android SDK.

En mayo de 2017, Google anunció que se agrega soporte para el lenguaje de programación Kotlin [28], un lenguaje moderno, de licencia abierta, e interoperable con Java, para facilitar el proceso de desarrollo de aplicaciones nativas [29].

Google provee el entorno de desarrollo Android Studio (figura 6), el cual está basado en el entorno IntelliJ Idea, de la empresa JetBrains. También es posible desarrollar desde cualquier otro entorno, pero Android Studio es el más utilizado debido a la gran cantidad de facilidades para el desarrollo que ofrece. Al descargarlo, también se incluye el Android SDK, emuladores, y una diversidad de plugins para acelerar el proceso de desarrollo. Permite previsualizar las pantallas de la aplicación en desarrollo, y armar las vistas usando la técnica de arrastrar y soltar. Está disponible de forma gratuita para Linux, Mac y Windows.

Debido a la naturaleza abierta de este sistema operativo, es utilizado por un gran número de fabricantes de dispositivos. En septiembre de 2017, el catálogo de dispositivos de Google Play contabilizaba 14.067 modelos diferentes. Esto plantea grandes desafíos para los desarrolladores, ya que deberán adaptar las aplicaciones para diferentes pantallas, idiomas, procesadores, versiones de Android, etc. Esto último es muy importante debido a que cada nueva versión de la API de Android va agregando avances. El desarrollador debe establecer una versión mínima de la API sobre la cual funcionará la aplicación. No se recomienda utilizar una API antigua, ya que no se estaría utilizando muchos de los avances de la plataforma, ni tampoco la última en el mercado, debido a que pocos serían los usuarios que podrían utilizarla (figura 7).

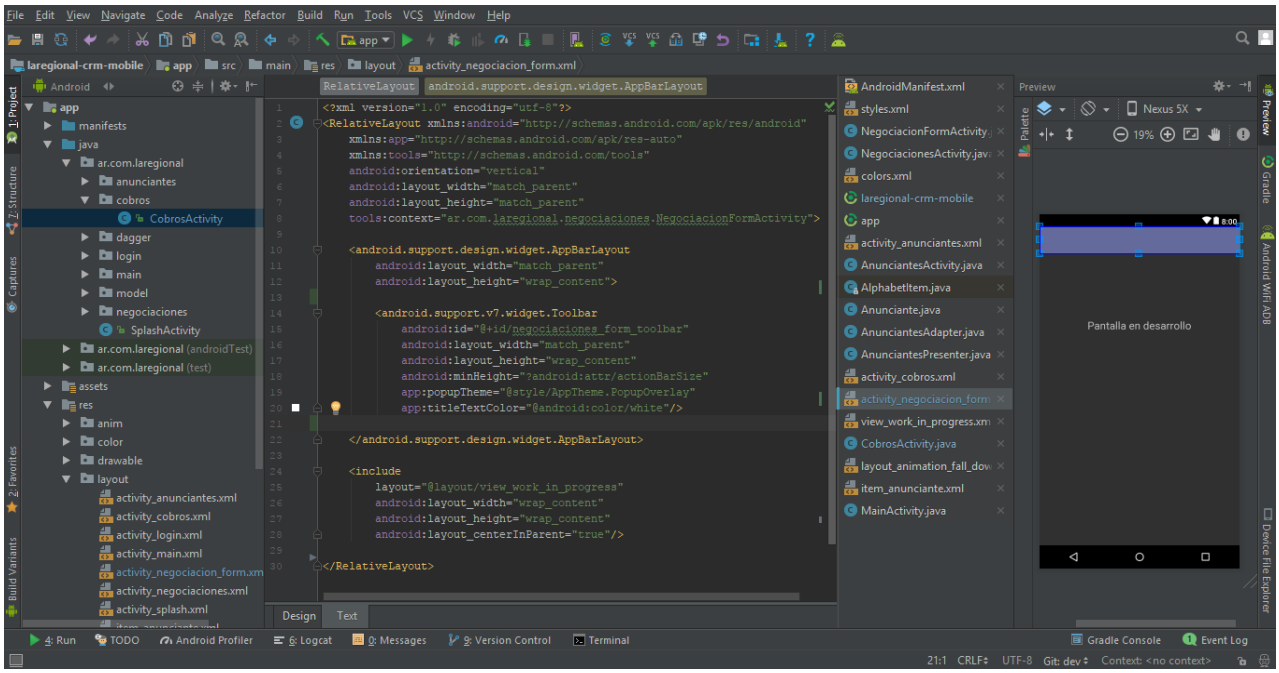
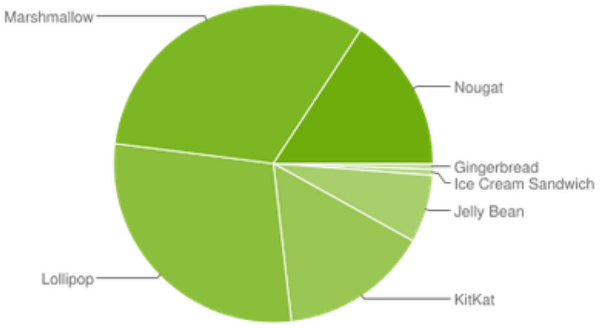


Figura 6. Entorno de desarrollo Android Studio

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.6%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.6%
4.1.x	Jelly Bean	16	2.4%
4.2.x		17	3.5%
4.3		18	1.0%
4.4	KitKat	19	15.1%
5.0	Lollipop	21	7.1%
5.1		22	21.7%
6.0	Marshmallow	23	32.2%
7.0	Nougat	24	14.2%
7.1		25	1.6%



Data collected during a 7-day period ending on September 11, 2017.
Any versions with less than 0.1% distribution are not shown.

Figura 7. Estadística de uso de versiones del sistema operativo Android

2.2.1.2 Desarrollo nativo en iOS

iOS es el sistema operativo desarrollado por Apple para sus teléfonos inteligentes iPhone y tabletas iPad. Es el segundo sistema operativo móvil más utilizado en el mundo.

Debido a que se ejecuta estrictamente en los modelos de iPhone y iPad de Apple, virtualmente no existen problemas de fragmentación como los que tiene Android.

En la figura 8 se aprecia que hay dos versiones principales de iOS en uso actualmente, 10 y 11 [30]. La versión 11 fue anunciada en junio de 2017, y en diciembre de 2017, como se ve en la figura 8 ya un 59% de los dispositivos iPhone usaban esta versión.

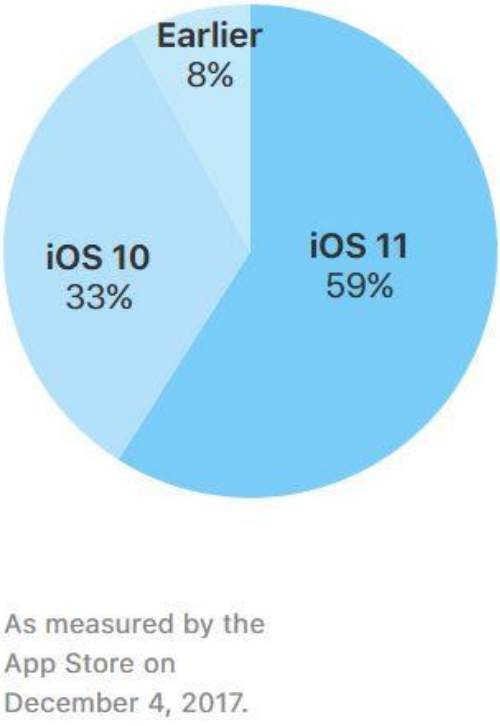


Figura 8. Estadística de uso de versiones del sistema operativo iOS

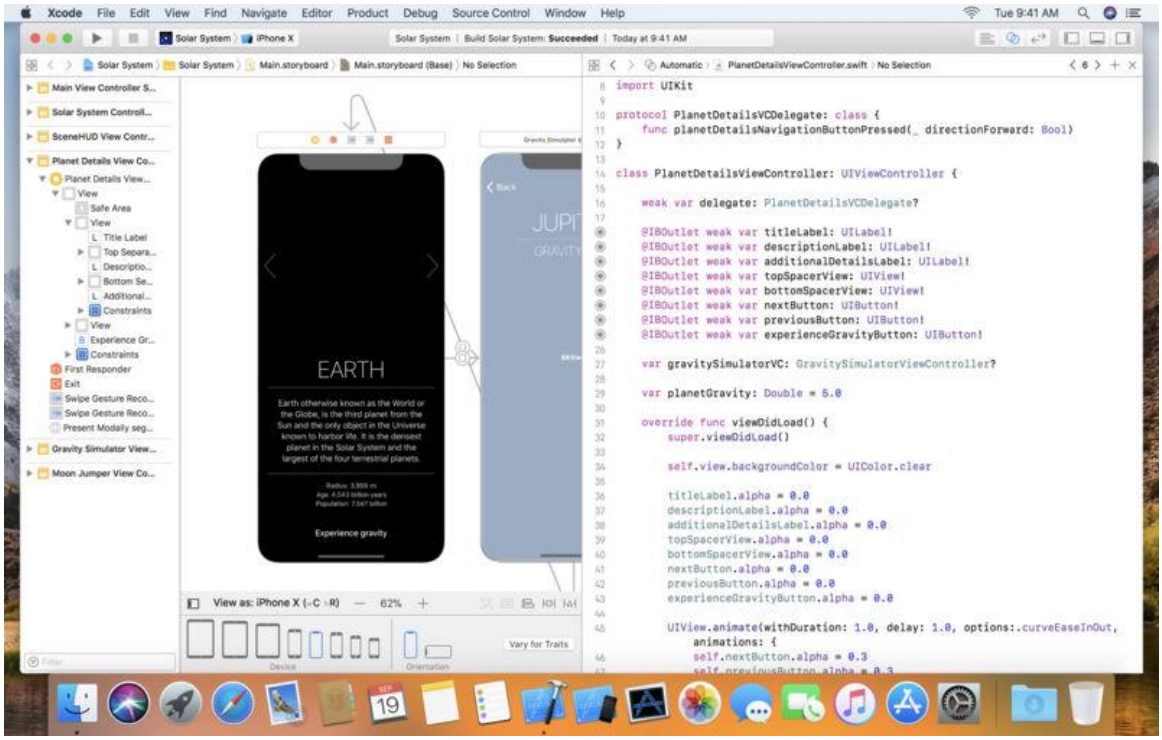


Figura 9. Entorno de desarrollo Xcode

La primera versión, presentada en 2007 se llamaba iPhone OS, y solo permitía aquellas aplicaciones que eran incluidas por Apple de forma estática, no había posibilidad de instalar otras ni eliminar las existentes. En 2008 se liberó el primer kit de desarrollo de aplicaciones nativas, usando el lenguaje Objective C. En 2014, se presentó el lenguaje Swift, desarrollado por Apple para facilitar el desarrollo de aplicaciones, tanto para iOS como para su sistema operativo para computadoras de escritorio, MacOS.

Para el desarrollo de las aplicaciones, se utiliza el entorno XCode (figura 9). Este provee muchas facilidades ya que incluye el framework iOS SDK, con el kernel XNU, Cocoa Touch, y las librerías de multimedia y demás servicios. Permite crear las interfaces gráficas con herramientas de arrastrar y soltar, además de las transiciones entre pantallas y la asociación de los componentes visuales nativos con el código en Swift.

Si bien el código fuente está compuesto por archivos de texto que podrían ser abiertos por otros programas, no es posible usar otros entornos de desarrollo para compilar y empaquetar las aplicaciones.

2.2.1.3 Desarrollo nativo en Windows Phone

Windows Phone es el sistema operativo móvil desarrollado por Microsoft. Fue lanzado en 2010, y es el tercero en cuanto a uso a nivel mundial, con una cuota muy reducida con respecto a los otros sistemas, de un 3% de los dispositivos [2].

Microsoft provee el IDE Visual Studio (figura 10), el cual está disponible para computadoras de escritorio con sistemas operativos Linux, Windows y Mac OS [31].

Se utiliza el lenguaje de programación C# con el framework .NET y el Windows SDK. También puede usarse C++.

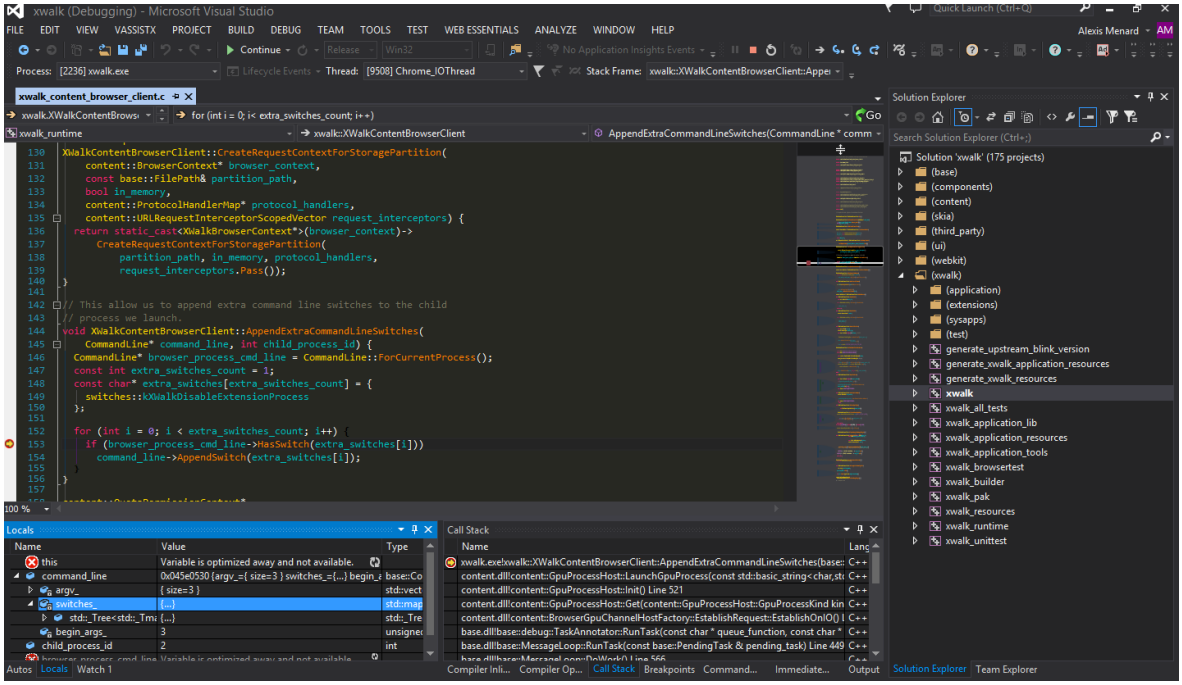


Figura 10. Entorno de desarrollo Visual Studio

2.2.2 Enfoque Web

El enfoque web consiste en desarrollar aplicaciones usando las tecnologías estándares web tales como HTML, CSS y JavaScript. Estas aplicaciones son accedidas desde todas las plataformas a través de navegadores web.

Para que esté accesible a los usuarios finales, se debe tener la aplicación preparada para ejecutarse en un servidor web que sea accesible a través de internet. Para acceder a la aplicación, el usuario debe ingresar a una URL pública desde su navegador web, que a través de HTTP solicitará la aplicación al servidor.

Esto hace que la puesta en producción y la distribución de actualizaciones sea instantánea, ya que todos los usuarios acceden al mismo servidor, pero trae aparejada la desventaja de que se debe tener acceso a internet para utilizarla.

La aplicación debe ser desarrollada usando técnicas para hacerla *responsive* o adaptable, es decir, que el diseño sea adaptable a los diferentes dispositivos y sus tamaños y tipos de pantalla (figura 11). Por ejemplo, un mismo sitio web puede adecuarse visualmente para ser visto en un teléfono inteligente, un televisor, una computadora de escritorio o incluso un reloj inteligente o un lector

de libros electrónicos (*e-reader*). Existen múltiples herramientas y frameworks para facilitar el desarrollo adaptable. Por ejemplo Bootstrap [32] es un framework que incluye componentes de HTML, CSS y JavaScript para acelerar el desarrollo de sitios orientados a móviles y adaptables.

Figura 11. Diseño web adaptable (responsive)

Una desventaja que aparece en este tipo de aplicación son las restricciones en el acceso al hardware del dispositivo. Las funcionalidades como el acceso a los sensores o almacenamiento en memoria persistente están muy limitados.

Las notificaciones Push, que son aquellas que llegan al dispositivo sin necesidad de que la aplicación esté abierta, pueden hacerse utilizando tecnologías como Web Push. [33] Con ésta, se utilizan estándares web tales como los workers de JavaScript [34], para que una aplicación web pueda recibir notificaciones como si fuera una aplicación nativa.

Si el usuario quiere guardar la aplicación para volver a usarla, debe guardar la URL de alguna forma, como por ejemplo, un marcador del navegador web.

Las aplicaciones web tradicionales no aparecen en el menú del sistema operativo como las demás aplicaciones. Para esto, existen tecnologías experimentales como las Progressive Web Apps [35] en Android, que permiten al usuario guardar parte de una aplicación web localmente para poder accederla desde su dispositivo sin necesidad de tener conexión a internet.

2.2.3 Enfoque Híbrido

En el enfoque híbrido, se busca combinar las ventajas del enfoque web con el nativo. Las aplicaciones híbridas están instaladas en el dispositivo, y el framework provee interacción con el hardware a través de APIs.

Se usan las tecnologías web estándar como en el enfoque web, pero estas se empaquetan dentro de una aplicación que consiste en un intérprete web, tal

como WebView en Android, donde se ejecuta el código HTML, CSS y JS. Este intérprete puede estar empaquetado dentro de la aplicación, tal como en el framework Crosswalk [36] o CocoonJS [37], o puede usarse el navegador web provisto por el sistema operativo, como en Apache Cordova [3].

La principal desventaja de este enfoque es que al no utilizarse los elementos visuales nativos para la interfaz, se puede perjudicar la experiencia de usuario, ya que los usuarios pueden no estar habituados a los componentes web, y además puede ralentizarse la ejecución por la carga asociada al contenedor web.

Para esto, se pueden integrar librerías de código abierto tales como jQuery [38], Materialize [39], entre muchas otras, que originalmente estaban pensadas para su uso en webs tradicionales, pero se fueron popularizando también en el desarrollo móvil híbrido. Ayudan a obtener una interfaz gráfica muy similar a las nativas, y así brindar una mejor experiencia de usuario.

2.2.3.1 Enfoque Híbrido con Apache Cordova

Cordova es un framework de código abierto desarrollado por la función Apache. Originalmente, fue desarrollado por la empresa Nitobi, hasta que Adobe la adquirió en 2011 y liberó una versión del código con licencia abierta. Adobe continúa el desarrollo de una versión comercial llamada PhoneGap, que usa como base a Cordova y agrega algunos servicios. También se han creado otros frameworks muy populares sobre Cordova, tales como Ionic [4], Intel XDK y CocoonJS [37].

Cordova permite desarrollar aplicaciones para Android, Blackberry, iOS, Windows, WebOS, FirefoxOS, entre varios otros.

La aplicación es desarrollada usando tecnologías estándares web como HTML5 y Javascript, y luego empaquetada en un container llamado WebView, que es nativo a cada plataforma.

Este WebView utiliza el navegador provisto por la plataforma. Esto trae la desventaja aparejada de que el código puede ser interpretado de maneras diferentes según el sistema operativo. Por ejemplo, en dispositivos iOS se usará el navegador web basado en Safari, y en Android dependerá de la versión. Esto se debe a que en versiones antiguas (menores a Android 4.4) se usaba el navegador web del sistema, y en versiones más recientes un WebView específico para aplicaciones con componentes web, basado en Chrome. Esto puede resultar en que, por ejemplo, algunos elementos visuales se interpreten de diferente forma según el dispositivo, o al utilizar diferentes motores de JavaScript, el mismo código se comporte de forma inesperada.

Una solución para esto es integrar con proyectos tales como Crosswalk, que está basado en Chromium, y permite empaquetar el intérprete web dentro de la aplicación. De esta manera, se logra que los componentes web se comporten exactamente igual en todas las plataformas. La desventaja que trae aparejada

es que al incorporar al navegador dentro del paquete de la aplicación, el tamaño de esta se incrementa en unos 20 megabytes, lo que puede llevar a que los usuarios no la instalen por falta de espacio.

2.2.3.2 Enfoque Híbrido con Ionic

Ionic [4] es un framework basado en Apache Cordova, y el framework JavaScript AngularJS. [40] Es de código abierto y fue presentado en 2013. Ganó popularidad rápidamente, llegando a crearse 1,3 millones de aplicaciones durante 2015. [41]

Ionic también provee Ionic Creator, un entorno de desarrollo que permite crear las vistas de la aplicación usando una interfaz visual con funcionalidades de arrastrar y soltar (*drag and drop*).

También permite utilizar TypeScript [42] junto con Angular, lo cual es muy atractivo para desarrolladores web que buscan aprovechar sus conocimientos en el campo de las aplicaciones móviles.

Al estar basado en Cordova, permite compilar las aplicaciones para todas las plataformas que este soporta.

2.2.3.3 Enfoque Híbrido con Sencha ExtJS

ExtJS es un framework para desarrollar aplicaciones multiplataforma usando estándares web, especializándose en interfaces táctiles. [43] Permite tener interfaces similares a las nativas en cada una de las plataformas. Para esto, incluye una gran variedad de componentes visuales listos para usar, que en la aplicación final tendrán distinta apariencia según la plataforma en la que se esté ejecutando.

Originalmente, se podía usar junto la herramienta Sencha Touch [44] de forma opcional, pero en 2016 Touch se incorporó a ExtJS, con el objetivo de ofrecer un paquete de herramientas completo.

Las aplicaciones desarrolladas con ExtJS se empaquetan usando Cordova, por lo que también se podrá llegar a todas las plataformas a las que llega este framework.

2.2.4 Enfoque Interpretado

En las aplicaciones interpretadas, se genera código nativo automáticamente para implementar la interfaz de usuario. Una parte del código desarrollado es traducido a código nativo para cada una de las plataformas a las que se apunta, mientras que la lógica de negocios es implementada independientemente usando diversas tecnologías como Javascript, Ruby, Java, XML, etc.

La principal ventaja de este enfoque es que el usuario interactúa con componentes visuales nativos de cada plataforma, obteniendo una buena experiencia de uso. Pero por otro lado, el código es fuertemente dependiente del framework que se utiliza. Por ejemplo, si una nueva versión de una plataforma agrega nuevas funcionalidades, como podrían ser nuevos componentes visuales en una versión de Android, estos estarán disponibles para la aplicación recién cuando el framework se actualice y los soporte.

Los frameworks más populares que usan este enfoque son NativeScript y Titanium.

2.2.4.1 Enfoque Interpretado con NativeScript

NativeScript [6] es un proyecto de código abierto que permite generar aplicaciones multiplataforma con interfaces nativas usando JavaScript. También es posible usar TypeScript, otro lenguaje de código abierto desarrollado por Microsoft, que se basa y es interoperable con JavaScript, agregando tipado estático y objetos basados en clases, entre otras características. Al momento de compilar la aplicación, el código TypeScript es traducido a JavaScript. Siendo que TypeScript es un superconjunto de JavaScript, el código escrito en este último es código válido en TypeScript.

Además, NativeScript permite usar Angular [40], un framework JavaScript muy popular que ayuda a acelerar el desarrollo. Por un lado, provee funcionalidades más avanzadas, y por otro, amplía las posibilidades de compartir código con proyectos web.

Según Telerik, la empresa encargada del desarrollo de NativeScript, se apunta a tener un 90% de reutilización del código entre las plataformas. [45]

NativeScript provee un sistema de plugins que agregan funcionalidades varias, como por ejemplo acceder al hardware del dispositivo. Además, permite reutilizar librerías específicas de Android e iOS, de Maven y CocoaPods respectivamente.

También provee un componente llamado *NativeScript Runtime*, que abstrae el hardware (cámara, GPS, sistema de archivos, etc.) y permite accederlo a través de una API de forma transparente a la plataforma.

Cuando la aplicación es compilada, parte del código multiplataforma es traducido a código nativo, mientras que el resto es interpretado en tiempo de ejecución. Los componentes visuales usan instancias de los componentes nativos de cada framework. Por ejemplo un botón será una instancia del objeto nativo Button de Android, y de UIButton en iOS. Al usar componentes nativos en vez de un intérprete web como en el enfoque híbrido, se logra una mejor performance y experiencia de usuario, ya que éste percibirá que la aplicación reacciona de forma instantánea a sus interacciones.

Según los creadores, la interpretación del código JavaScript y los movimientos de datos agregados por las diferentes capas de NativeScript tienen un impacto

en la performance de un 10%, comparado con la misma implementación con las herramientas nativas. [46]

2.2.4.2 Enfoque Interpretado con Appcelerator Titanium

Appcelerator Titanium [5] es un framework de código abierto que permite desarrollar aplicaciones para iOS, Android, Windows, Blackberry y la web con HTML5.

Este framework está conformado por Titanium, la herramienta para generar aplicaciones multiplataforma partiendo de una base de código en común JavaScript. Y por otro lado, está Alloy, un framework que se agrega a Titanium para organizar el código siguiendo el patrón MVC, con soporte para herramientas de desarrollo tales como el framework Node.js, UnderscoreJS, Backbone.js, y la posibilidad de crear vistas en XML para separar la interfaz visual y los estilos de la lógica de negocios. Además, Appcelerator, la empresa detrás de Titanium, ofrece Hyperloop, [47] una herramienta propietaria que permite acceder a las APIs nativas de Android, iOS y Windows.

Las aplicaciones se desarrollan usando código JavaScript, y éste se interpreta en tiempo de ejecución de la aplicación a través de un intérprete.

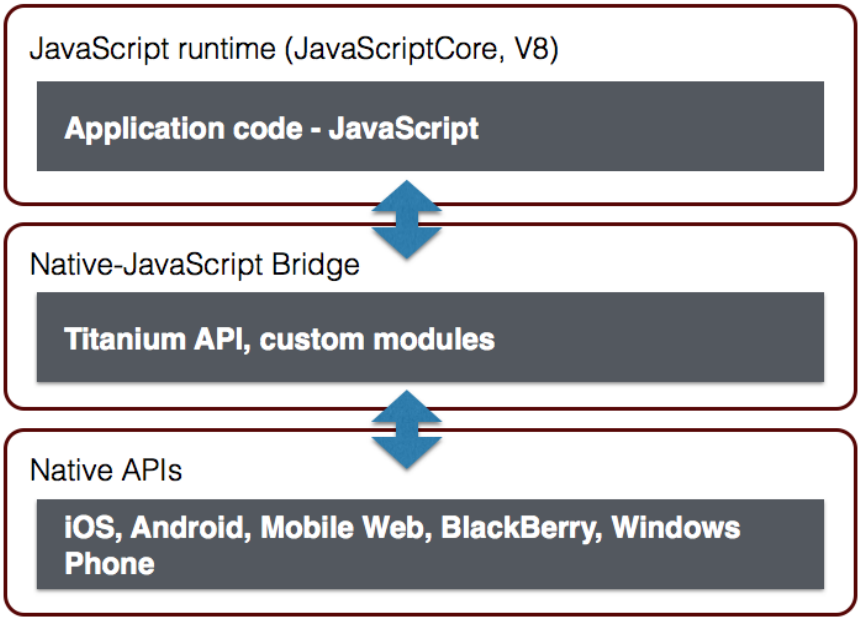


Figura 12. Arquitectura de aplicaciones creadas con Titanium

La API ofrecida por Titanium se encarga de mapear los elementos referenciados en JavaScript uno a uno a elementos nativos para realizar acciones como dibujar botones, abrir ventanas, tomar una foto, etc. Al mismo tiempo, el SDK se encarga de que todos los eventos que ocurren en el entorno nativo sean mapeados a JavaScript para ser manejados por el código multiplataforma.

Los creadores de Titanium destacan que mediante este framework es posible reutilizar entre el 60% y 90% del código entre distintas plataformas.

2.2.5 Enfoque de aplicaciones generadas por compilación cruzada

En el enfoque de aplicaciones generadas por compilación cruzada, la aplicación es desarrollada utilizando una base de código en común, que es traducida a código nativo para cada plataforma. Gracias a esto, se consigue teóricamente una mejor performance, y acceso total a las APIs del hardware del dispositivo. Teóricamente, también se puede modificar el código generado para optimizarlo, ajustar algún requerimiento específico, o incluso para adoptar nuevas funcionalidades que la plataforma de destino provea al actualizarse. Sin embargo, en la práctica la utilización del código generado es difícil debido a que el código tiene una estructura automatizada que puede ser difícil de comprender.

Por ejemplo, el framework Applause [48] [49] define un lenguaje de dominio específico o DSL (del inglés *Domain-specific Language*), que se usa para describir aplicaciones móviles orientadas a datos, y luego se usan generadores de código nativo para generar versiones para iOS, Android, o Windows Phone. La lógica de negocio se comparte entre todas las plataformas, escrita en JavaScript.

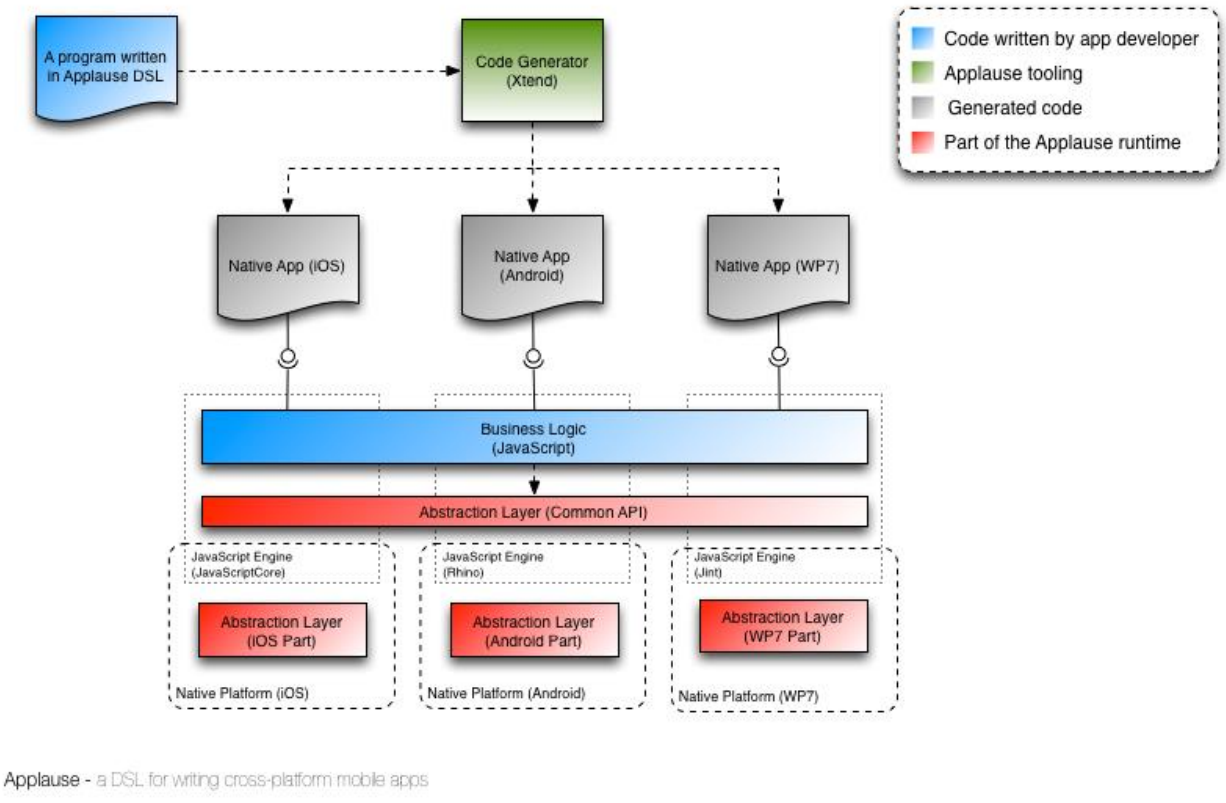


Figura 13. Arquitectura de aplicaciones generadas con Applause

Frameworks como Applause usan un enfoque de desarrollo orientado a modelos. Se usa un lenguaje de modelado para representar la funcionalidad y el comportamiento de la aplicación. Luego, el proceso de generación del código de la aplicación se hace en base a ese modelo. Esto implica que el desarrollo de las aplicaciones se haga con un alto nivel de abstracción. Sin embargo, proyectos de código abierto con este enfoque como Applause o iPhonical [50] no han tenido éxito, y finalmente fueron abandonados.

Por otro lado existen frameworks que siguen el enfoque de compilación cruzada y han ganado mucha popularidad. Por ejemplo Xamarin y RubyMotion, generan aplicaciones nativas compartiendo una base de código entre las plataformas, C# y Ruby respectivamente.

2.2.5.1 Enfoque de aplicaciones generadas por compilación cruzada con Corona

Corona es un framework multiplataforma que permite desarrollar tanto aplicaciones de propósito general como juegos, para las plataformas más populares del mercado, incluyendo macOS, Windows, iOS, Android, Kindle, Apple TV, Android TV y Steam.

Se usa un único código base, desarrollado usando el lenguaje de scripting Lua, sin necesidad de escribir código específico para ninguna de las plataformas. Esto permite tener una reutilización del código total, con la desventaja de que las interfaces generadas no serán similares a las de las aplicaciones nativas. [51]

Corona es un framework de código cerrado. Al iniciar una compilación, se pre-compila el código Lua a bytecode, y se lo envía al servidor web de CoronaLabs, la empresa creadora del framework. El servidor integra el código recibido con el motor de Corona correspondiente a la plataforma de destino, y finalmente se descarga el paquete ejecutable de lado del usuario. [52]

Su objetivo principal es la construcción de juegos multiplataforma de manera simple y rápida, haciendo las diferencias entre plataformas transparentes para el programador. También provee plugins y acceso a APIs nativas.

2.2.5.2 Enfoque de aplicaciones generadas por Compilación cruzada con Xamarin

Xamarin es un framework para crear aplicaciones multiplataforma con interfaces nativas, con una reutilización de código de entre 75% y 100%. [53] Se usa una base de código C# con el framework .NET, en común para la lógica de negocio y los controles de la interfaz de usuario. Los componentes visuales se programan con las herramientas nativas de cada plataforma.

También provee la herramienta de generación de vistas llamada Xamarin.Forms, que en base a una única definición, genera las interfaces de

usuario nativas para todas las plataformas, de esta forma alcanzando el 100% de reutilización del código.

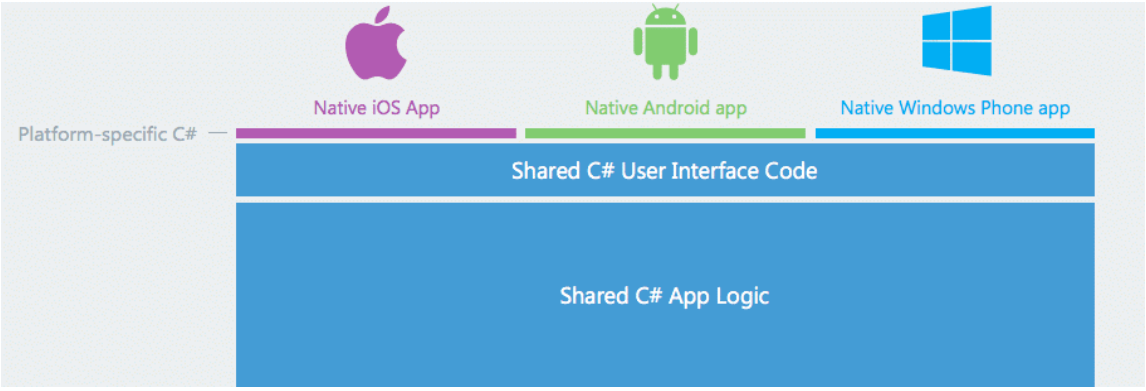


Figura 14. Arquitectura de una aplicación desarrollada con Xamarin

Al momento de la compilación, el código C# es transformado según la plataforma de destino. Las clases que quedan sin uso se eliminan para reducir el tamaño del paquete final. [54]

Para iOS, es compilado a lenguaje de máquina ARM, junto con el framework .NET. Debido a las limitaciones impuestas por iOS, el código se compila estáticamente antes de la ejecución, se limitan algunas funcionalidades como por ejemplo los tipos genéricos y la generación de código dinámicamente con reflexión.

En el caso de Android y Windows, el código es compilado al lenguaje intermedio de C#. En Android, se empaqueta junto con el intérprete MonoVM, y la aplicación se ejecuta junto con Android Runtime, interactuando con los tipos nativos usando JNI (*Java Native Interface*). En Windows, se ejecuta de forma nativa con el intérprete provisto por la plataforma.

2.3 Desafíos para la Ingeniería de Software

Hace apenas unas décadas atrás, el uso de los sistemas de software estaba restringido a un grupo reducido de usuarios especializados. Aquella época contrasta fuertemente con la actualidad en la que los teléfonos inteligentes, pequeñas computadoras móviles de propósito general, se han vuelto cotidianos y omnipresentes. Estos dispositivos pueden utilizarse para llevar a cabo tareas complejas y críticas, requiriendo la mejora continua de las capacidades de cómputo, la disponibilidad, el rendimiento eficiente, entre otras necesidades. La evolución vertiginosa de esta tecnología ejerce presión en la adaptación de la Ingeniería de Software.

Respecto del desarrollo para dispositivos móviles se debe considerar una serie de características propias de esta actividad que no estaban presentes en el desarrollo tradicional de software. [55] Es necesario tener en cuenta sobre qué tipo de dispositivos se debe ejecutar la aplicación a construir. La diversidad de plataformas, lenguajes de programación, herramientas de desarrollo, estándares, protocolos y tecnologías de red; algunas limitaciones de ciertos dispositivos y las exigencias de tiempo del mercado, entre otros, constituyen algunos de los problemas a tratar.

En la mayoría de los casos, el éxito de un producto de software para dispositivos móviles estará condicionado por el nivel de popularidad que logre alcanzar. Para maximizar su presencia en el mercado es necesario que la aplicación pueda ejecutarse sobre la mayor cantidad de plataformas móviles existentes, especialmente sobre las dos más populares: Android e iOS. [56] Para lograr este objetivo existen dos alternativas:

i) Desarrollar aplicaciones específicas para cada plataforma, impulsando en paralelo varios proyectos de desarrollo, con las herramientas y lenguajes propios de cada una de ellas. Estas aplicaciones, analizadas previamente en este trabajo, se denominan aplicaciones nativas.

ii) Generar aplicaciones que puedan ejecutarse en forma directa sobre más de una plataforma de sistema operativo a partir de un único proyecto de desarrollo. Estas aplicaciones se denominan aplicaciones multiplataforma.

En los últimos años se ha incrementado el interés de la comunidad de Ingeniería de Software por el estudio del desarrollo de aplicaciones multiplataforma para dispositivos móviles.

Atendiendo a este creciente interés, en [7] los autores presentaron un estudio comparativo entre las diferentes estrategias de desarrollo de aplicaciones. Se siguió la taxonomía de aplicaciones nativas, web, interpretadas, híbridas, y generadas por compilación cruzada, analizando las ventajas y desventajas de cada enfoque. Se usaron como criterios de comparación la posibilidad de distribución por las tiendas de aplicaciones, la popularidad de las tecnologías empleadas, el acceso al hardware y datos, el *look and feel* de la interfaz de usuario, y el rendimiento percibido por los usuarios. Se concluye que los

usuarios notan mejor experiencia en las aplicaciones nativas, así como la interfaz resultante, que se integra mejor con la apariencia del resto de las aplicaciones de la plataforma.

Sin embargo, se menciona que algunos frameworks como Titanium y Cordova resultan muy accesibles de entender para los programadores debido al uso de tecnologías estándares web, acelerando el desarrollo; y en el caso de Titanium, logrando una interfaz nativa. Queda en claro que con la evolución de estas herramientas, las aplicaciones híbridas e interpretadas tendrán un rol fundamental en el futuro de la Ingeniería de Software móvil. Los autores proponen como trabajo futuro continuar el análisis de los frameworks, sumando parámetros que afecten a los requerimientos no funcionales, como la performance, tiempo de inicio, tiempos de ejecución, etc.

En [57] se hace un estudio comparativo de frameworks de desarrollo de aplicaciones multiplataforma, haciendo énfasis en la performance de aplicaciones cuya funcionalidad requiera un alto nivel de uso del procesador. Para esto, se sigue la taxonomía propuesta en [7], creando aplicaciones en todos los enfoques de desarrollo, es decir, nativo, web, híbrido, interpretado y generado por compilación cruzada. Se midió el tiempo de ejecución de los algoritmos implementados con cada uno de los enfoques, concluyendo que las aplicaciones multiplataforma pueden potencialmente tener muy buena performance, incluso en algunos casos más que las aplicaciones nativas. Se observa que las aplicaciones web también tuvieron buenos resultados, más allá del sistema subyacente.
Los autores subrayan que hoy en día, al encarar un desarrollo de software, es posible crear la versión para dispositivos móviles, aunque esto no es una tarea simple. La decisión del método de desarrollo resulta fundamental, y este tipo de comparativas puede ser útil para los ingenieros de software al momento de elegir el enfoque de desarrollo de la aplicación móvil. Sin embargo, es muy limitada la cantidad de artículos de otros autores que analicen la performance de los diferentes enfoques de desarrollo de aplicaciones móviles, dado que se hace mucho énfasis en los enfoques nativo e híbrido.

Capítulo 3. Experimentación

Con el objetivo de estudiar y comparar las diferentes metodologías de desarrollo de aplicaciones móviles, se llevaron a cabo varios experimentos en simultáneo. Se hizo énfasis en varios aspectos que pueden resultar determinantes para el éxito de una aplicación, mencionados anteriormente.

Los tres primeros experimentos consistieron en analizar el consumo de batería usando cada uno de los enfoques de desarrollo descriptos. El siguiente experimento, tomó en cuenta otro factor de los dispositivos móviles, el escaso espacio de almacenamiento persistente. Para este experimento, se analizó el tamaño de las aplicaciones creadas, implementado la misma funcionalidad con los diferentes enfoques.

3.1 Descripción del problema

3.1.1 Consumo de batería

El impacto que la aplicación tendrá sobre la batería del usuario resulta un factor fundamental que debe ser tenido en cuenta al momento de la elección del modo de desarrollo.

Una aplicación que consuma rápidamente la energía de la batería puede llevar a que los usuarios la desinstalen, sin importar el trabajo que pueda haberse hecho en otras partes respecto a la experiencia de usuario. También puede llevar a que tenga malas calificaciones en las tiendas de aplicaciones, lo cual para los desarrolladores, resulta extremadamente importante de evitar. [58] Por ejemplo, aplicaciones de redes sociales masivas han recibido malas calificaciones, haciendo que los usuarios recomienden desinstalar la aplicación y utilizar el sitio web como alternativa. [59]

Existen muchos tipos de mecanismos de administración de energía que buscan reducir el consumo. Suelen estar implementados a nivel de sistema operativo, diseñados para operar sin deteriorar o penalizar la performance de las aplicaciones y la experiencia de usuario, siendo invisibles para los desarrolladores de aplicaciones. [60] Algunos ejemplos son el voltaje y frecuencia dinámicos de los procesadores, que aumentan cuando la aplicación está realizando una actividad muy demandante de procesamiento y luego bajan en momentos de inactividad, o los modos de ahorro de energía de las antenas, quienes entran en estado de suspensión cuando no se transmiten datos.

Reducir la potencia no necesariamente implica reducir el consumo de energía. Por ejemplo, disminuir la potencia del procesador bajando la velocidad del reloj, podría oponerse a la eficiencia energética, dando lugar a un tiempo de ejecución más largo que podría incrementar el consumo total de energía. Los dispositivos de baja potencia que funcionan durante mucho tiempo podrían

consumir más energía que los dispositivos de mayor potencia que funcionan durante un tiempo más corto. [61]

Muchos estudios han mostrado que la eficiencia de una aplicación puede ser mejorada a través de cambios en el código, más allá de los mecanismos disponibles a nivel sistema operativo. Por ejemplo, [62] muestra que el consumo energético de aplicaciones de streaming multimedia puede ser reducido drásticamente al implementar un algoritmo para planificar la transmisión y recepción de los datos, reduciendo así el uso de las antenas y por ende el consumo de batería.

En [1] y [63] se presentaron recomendaciones para el desarrollo de aplicaciones con el objetivo de reducir el consumo de energía, desde el punto de vista del software. Entre estas recomendaciones se encuentran agrupar el código según la prioridad, para que pueda ser ejecutado con diferentes frecuencias de procesador según la disponibilidad de energía, reducir el uso de rutinas que se ejecuten periódicamente en el tiempo, maximizar el uso de la multiprogramación y las cachés, entre otras.

En [64] los autores del artículo concluyeron que la mayoría (5 de 8) de las mejores prácticas de programación publicadas por Google para optimizar el rendimiento de aplicaciones Android, también impactan positivamente sobre el consumo de energía.

Una razón por la cual una funcionalidad puede ser implementada de manera energéticamente ineficiente, es que el origen del problema puede ser difícil de detectar. Los problemas de la ejecución podrían no tener ningún otro síntoma visible más que gastar la energía de la batería silenciosamente. La comunidad suele referirse a esto como un “*bug* de energía”. [65]

Por todo esto, es muy importante que los desarrolladores estén al tanto del comportamiento de la aplicación en cuanto a la energía utilizada, y el impacto que tendrá el framework de desarrollo elegido sobre ésta. El problema es que la mayoría de los dispositivos no provee una API que reporte el consumo instantáneo.

Para esto existen diferentes métodos de tomar mediciones, tanto por software como por dispositivos físicos.

Entre los métodos físicos, uno popular y con mucha precisión es usar un medidor externo tal como el *Monsoon Power Monitor* [66], pero esto requiere desarmar físicamente el dispositivo y conectar el medidor directamente a la batería tal como se muestra en la figura 15. Debido a la naturaleza compacta de los teléfonos inteligentes actuales, esto puede ser un problema ya que en muchos modelos no puede accederse fácilmente a la batería. Además, representa grandes costos extras, ya que se trata de un tipo de equipamiento que si bien es viable para un laboratorio de investigación, no es común entre los desarrolladores de aplicaciones.

Debe considerarse además que los valores obtenidos con unidades de medición externas conectadas a la interface de la batería, indican el consumo de energía total del dispositivo. No resulta fácil identificar cuáles son los recursos de hardware, utilizados por la aplicación monitoreada que están contribuyendo al consumo total de energía. Aunque es posible extraer las contribuciones de algunos subcomponentes individuales a través de un análisis extenso de los datos de medición, el esfuerzo requiere conocimiento específico del dominio y no siempre es factible realizarlo. [67]

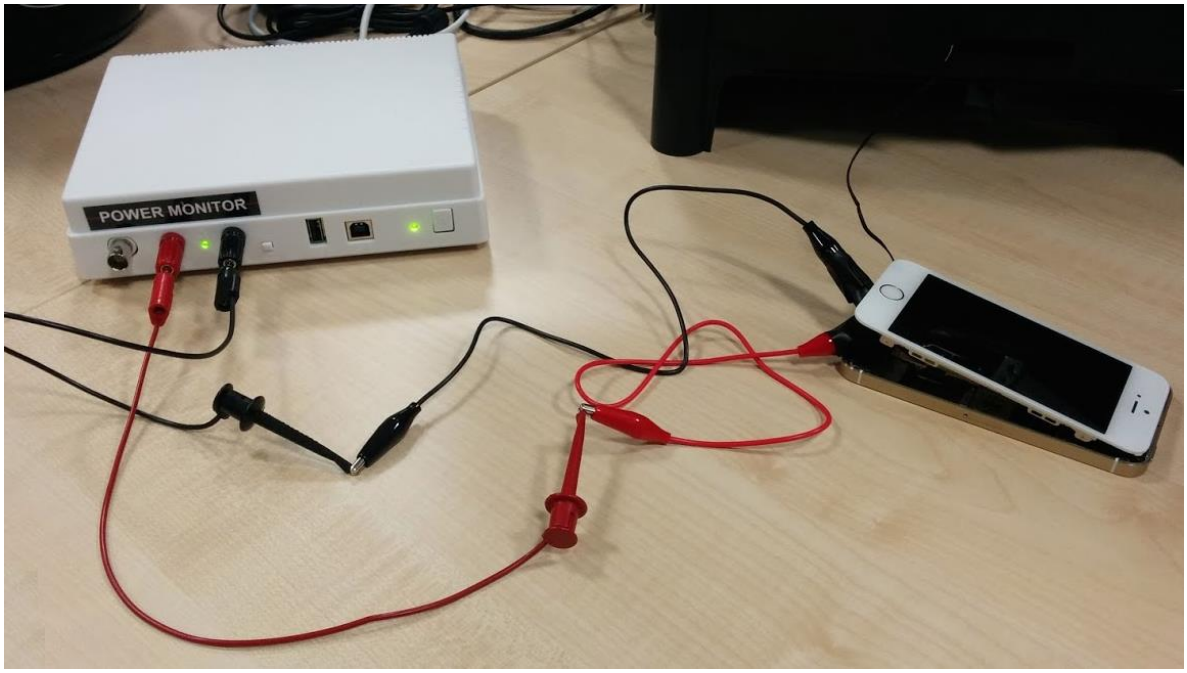


Figura 15. Un smartphone desarmado y conectado a un medidor de potencia Monsoon.

Otros métodos por software buscan aprovechar las APIs del sistema operativo para mostrar la información que pueda recabarse, indicando el consumo de cada aplicación, de todo el dispositivo, o de cada componente de hardware, tal como pantalla, CPU, etc.

Estos métodos por software resultan más amigables para los desarrolladores de software, ya que no requieren de equipamiento extra.

El análisis sobre el consumo de energía se llama *energy profiling*, o perfilamiento de energía, y las aplicaciones utilizadas para realizar este análisis reciben el nombre de *energy profilers*. Una funcionalidad básica de estos *energy profilers* es proporcionar información sobre el uso de la batería tal como la energía consumida por todo el dispositivo o discriminada por componente de hardware y/o aplicación. Esta metodología no requiere la conexión física de ninguna herramienta de medición y por ello resulta más sencilla de utilizar que los métodos que involucran dispositivos como multímetros externos. Las estimaciones sobre el consumo de energía suelen estar basadas en modelos de potencia.

Un modelo de potencia es una función matemática que cuantifica los factores que impactan sobre el consumo de energía, como la utilización de un determinado subcomponente de hardware. Como ejemplo de un modelo de potencia simple de grano grueso considérese uno que estima el consumo del sistema como una función de una sola variable: el nivel de brillo. [67] El sistema operativo Android tiene sus propios modelos de energía y profilers para estimar el consumo de energía de diferentes componentes y aplicaciones. Un ejemplo de una herramienta de software basada en un modelo de potencia de grano fino que permite estimar el consumo energético de una determinada aplicación es "eprof" [68] que fue implementada para los sistemas operativos Android y Windows Mobile.

Otra alternativa para analizar el consumo de energía en dispositivos móviles combina la practicidad del software de estimación de consumo y la precisión de las mediciones reales. Este método depende de ciertas facilidades que proveen algunos fabricantes de hardware que incorporan sensores en los circuitos internos de sus dispositivos. Así es posible monitorear el consumo energético en tiempo real de cada subcomponente por medio de un *profiler* adecuado. Ejemplo de esto son los procesadores Snapdragon de Qualcomm Technologies, Inc. junto con la aplicación Trepn Profiler [69] desarrollada por la misma compañía. El uso de estas herramientas ha ganado popularidad entre los desarrolladores de aplicaciones y también ha sido utilizada en estudios de investigación publicados como por ejemplo en [70], donde se lo utilizó para obtener métricas de ejecución de diferentes aplicaciones.

3.1.2 Uso de espacio de almacenamiento

El uso del espacio de almacenamiento, resulta fundamental que sea tenido en cuenta al momento de desarrollar una aplicación. La utilización excesiva de almacenamiento puede implicar que los usuarios eliminen la aplicación, o directamente decidan no instalarla.

El mercado actual de teléfonos inteligentes se compone de una mayoría de dispositivos con almacenamiento muy limitado. Al espacio declarado por el fabricante en las especificaciones, se le debe deducir el espacio ocupado por el sistema operativo y las aplicaciones instaladas de fábrica.

En [71] y [72] se proponen modelos de diseño de aplicaciones móviles elásticas, integrando el concepto de *cloud computing*. De esta forma, se busca aumentar los recursos de cómputo y almacenamiento limitados de los dispositivos móviles, separando la aplicación en módulos, y llevando a la nube aquellos módulos que sean más demandantes de recursos.

Por otro lado, el almacenamiento persistente es el componente de memoria más lento en los dispositivos móviles. Dado que tiene un rol muy importante en los sistemas móviles, puede afectar en gran manera a la experiencia de usuario.

Además de las desventajas evidentes cuando una aplicación usa mucho espacio, esto también puede traducirse en impacto sobre el consumo de energía. [73] Varios trabajos han mostrado cómo el uso del almacenamiento impacta directamente sobre la batería de los dispositivos.

En [74] se discute cómo grandes lecturas y escrituras en el almacenamiento de los dispositivos móviles consumen energía, y se presentan métodos para reducir este consumo.

En [75] se muestra que el uso de medios de almacenamiento con gran tamaño pero bajo costo, como pueden ser las tarjetas de memoria externas, utilizan más energía. Esto se debe a que el uso de almacenamiento “lento” implica más uso de CPU durante las operaciones de entrada/salida.

La mayoría de los frameworks de desarrollo consideran de algún modo la problemática del tamaño final de las aplicaciones, intentando reducirlo.

En el caso de las aplicaciones en Android usando el enfoque nativo, está disponible la herramienta ProGuard. Esta aprovecha el beneficio de los lenguajes con tipado estático, y hace un análisis de todo el código al momento de la compilación. De esta forma, detecta cuáles partes del código y cuáles recursos no van a ser utilizados durante la ejecución, y los elimina (clases, métodos y recursos externos como imágenes).

El framework Xamarin también provee una herramienta de análisis estático del código para eliminar partes que no serán ejecutadas. Al crear aplicaciones para Android, dentro del paquete de la aplicación se incluye el intérprete de código C# MonoVM. La herramienta analiza este intérprete, que de base ocuparía

aproximadamente 13MB, y busca reducir su tamaño al mínimo posible sin comprometer las funcionalidades de la aplicación.

Al generar aplicaciones para Windows Phone, en cambio, no hará falta empaquetar el intérprete ya que todo el código será ejecutado por el *runtime* provisto por el sistema operativo.

En el caso de los frameworks que utilizan lenguajes con tipado dinámico, como JavaScript, se encaran las optimizaciones del almacenamiento utilizado con otras estrategias. Entre estas se encuentran la eliminación de partes sin utilizar de los intérpretes, la separación de los paquetes finales en varios, cada uno específico para una arquitectura de hardware, o la *minificación* del código, una técnica automatizada que consiste en reducir el tamaño del código eliminando todos los espacios en blanco y comentarios, y uniendo varios archivos en uno.

En NativeScript, cada aplicación es empaquetada junto con la máquina virtual de JavaScript, V8, una capa de código C++ que comunica a la VM de JavaScript con las interfaces nativas de Android y el hardware, y una capa en Java encargada de comunicar las diferentes partes. [76] Las dos primeras partes son código compilado y dependen de la arquitectura subyacente, a diferencia de Java y Javascript que funcionan independientemente del hardware.

Esto implica que debe haber una versión diferente de estas capas para cada arquitectura soportada por el sistema operativo Android, lo que resulta que el tamaño base de una aplicación desarrollada con NativeScript será de aproximandamente 11MB. Este problema no existe al generar aplicaciones para la plataforma iOS, ya que debido a la naturaleza cerrada de este “ecosistema”, todos los dispositivos usan la misma arquitectura de cómputo.

La solución que plantea la documentación de NativeScript es aprovechar la funcionalidad de “múltiples APK” que provee la tienda de aplicaciones Google Play Store. [77] Esta consiste en generar individualmente un paquete APK para cada arquitectura, en lugar de incluir todos los archivos binarios de las diferentes arquitecturas en un mismo paquete. Esto implica mucho más trabajo para el desarrollador, ya que para cada arquitectura, deberá generar la aplicación y eliminar los binarios de las demás arquitecturas, y luego agregar individualmente los archivos APK resultantes a la tienda de aplicaciones.

La documentación de Titanium propone varias optimizaciones, algunas relacionadas con los recursos, y otras con los intérpretes compilados para arquitecturas específicas. [78] Por el lado de los recursos, plantea separar todos los recursos como imágenes, gráficos, audio y elementos de la interfaz gráfica en diferentes directorios, específicos para cada plataforma de destino; de esta forma se evitará empaquetar archivos que no serán utilizados.

Otro enfoque que busca reducir la descarga inicial de contenidos es el de “archivos de expansión” que provee la tienda de aplicaciones Google Play Store. [79] Este permite empaquetar recursos de gran tamaño, y solo

descargarlos cuando el usuario lo desee, lo cual se hace de forma transparente gracias a que es un servicio incorporado en el sistema operativo.

Por el lado de las arquitecturas, en Titanium ocurre una situación similar a la descrita en NativeScript, donde el intérprete de JavaScript es un binario individual para cada arquitectura. Titanium permite compilar específicamente para cada arquitectura, de forma más sencilla que NativeScript, y luego es decisión del desarrollador utilizar la funcionalidad de múltiples APK de la tienda de aplicaciones.

También se proponen otras técnicas que son válidas más allá del framework utilizado.

Una es generar elementos de la interfaz programáticamente usando vectores, evitando tener archivos de imagen que representen, por ejemplo, botones o flechas.

Otra es el concepto de *offloading* [80], es decir, llevar los recursos más pesados a un servidor web, descargándolos bajo demanda, al momento de usarlos.
En el caso del framework del enfoque híbrido Cordova, el código de la aplicación es ejecutado por el navegador web provisto por el sistema operativo. Esto se traduce en aplicaciones comparativamente muy livianas, donde los elementos más influyentes en el tamaño final serán el código HTML y JavaScript, y los recursos multimediales incluidos.

La mayoría de las discusiones sobre optimización de uso del espacio en Cordova y los frameworks derivados gravitan en torno a optimizar los recursos multimedia. Como se mencionó anteriormente, cuando se quiere incluir un intérprete o navegador web dentro de la aplicación, es posible gracias a proyectos de código abierto como Crosswalk. Con esto se logra tener un comportamiento similar en todas las plataformas. Sin embargo, la desventaja es el aumento en el tamaño de la aplicación. La documentación de Crosswalk [81] aclara que se agregarán unos 20 megabytes al tamaño total. Durante la realización de los experimentos de este trabajo, se hizo la prueba de agregar Crosswalk a las aplicaciones desarrolladas con Cordova, y el incremento fue de 26,94 megabytes.

3.2 Diseño de los experimentos

Las pruebas cuyos resultados se presentan en este trabajo fueron realizadas sobre la plataforma Android, dado que en la actualidad representa la mayor parte del mercado mundial de teléfonos inteligentes.

Android es una plataforma abierta, por lo que el proceso de medición puede ser más controlado y detallado en comparación con otras plataformas como iOS y Windows Phone. Sin embargo, es importante también hacer foco sobre otras plataformas como iOS y Windows Phone que se deja planteado como trabajo futuro.

Para llevar a cabo la experimentación, se eligieron los frameworks de desarrollo multiplataforma según los diferentes enfoques presentados anteriormente. Se diseñaron pruebas para evaluar el comportamiento de las aplicaciones usando:

1. Android Nativo (NDK y Java)
2. Apache Cordova (multiplataforma híbrido)
3. Appcelerator Titanium (multiplataforma interpretado)
4. NativeScript (multiplataforma interpretado)
5. Xamarin (multiplataforma compilación cruzada)
6. Corona (multiplataforma compilación cruzada)

Se desarrollaron aplicaciones con idéntica funcionalidad en cada uno de los frameworks elegidos.

Para los experimentos de evaluación de consumo de batería, se desarrollaron aplicaciones de reproducción de audio, de video, y de procesamiento intensivo. Para esta última, se desarrolló también una versión usando el NDK de Android.

Para evaluar el uso de espacio de almacenamiento, se usaron las aplicaciones de reproducción de audio, video y se agregaron aplicaciones basadas en texto.

De esta manera, quedaron definidos 19 casos de prueba para los experimentos de evaluación de consumo de batería, y 16 para el uso de espacio de almacenamiento.

El código fuente de todos los desarrollos realizados se encuentra accesible públicamente en. [82]

Las diferencias observadas en un conjunto de pruebas preliminares sobre dispositivos móviles de distintas marcas y modelos no resultaron demasiado significativas. Se decidió entonces basar toda la experimentación de esta investigación en un smartphone presente en el mercado al momento de llevar a cabo este estudio, marca Motorola, modelo Moto-G2, procesador Quad-core 1.2 GHz Qualcomm Snapdragon 400, GPU Adreno 305, 1GB de RAM, y sistema operativo Android 6.0. Este dispositivo surgió como un buen

representante promedio de todos los demás examinados en la fase de pruebas preliminares.

3.2.1 Consumo de batería

Para el cálculo de la energía consumida por cada ejecución de las aplicaciones, se buscaron y analizaron diferentes herramientas.

Con la finalidad de tener un mayor control y precisión sobre los resultados, se decidió utilizar un profiler desarrollado por el fabricante del hardware que aprovecha los sensores internos que colocan en sus propios componentes. Es por ello que para realizar las mediciones del consumo de energía se eligió la herramienta Trepn Profiler de Qualcomm, la misma compañía que desarrolló el procesador Snapdragon, del smartphone utilizado en las pruebas.

La herramienta se encuentra bien documentada [69] y ha alcanzado gran popularidad en la tienda de aplicaciones Google Play. [83] Esta herramienta tiene una buena precisión ya que toma los datos del circuito integrado de administración de energía de los procesadores Snapdragon que se detalla en la figura 16 (PMIC por sus siglas en inglés).

En [67] se compara a Trepn con otros sistemas de medición, concluyendo que tiene una precisión del 99%.

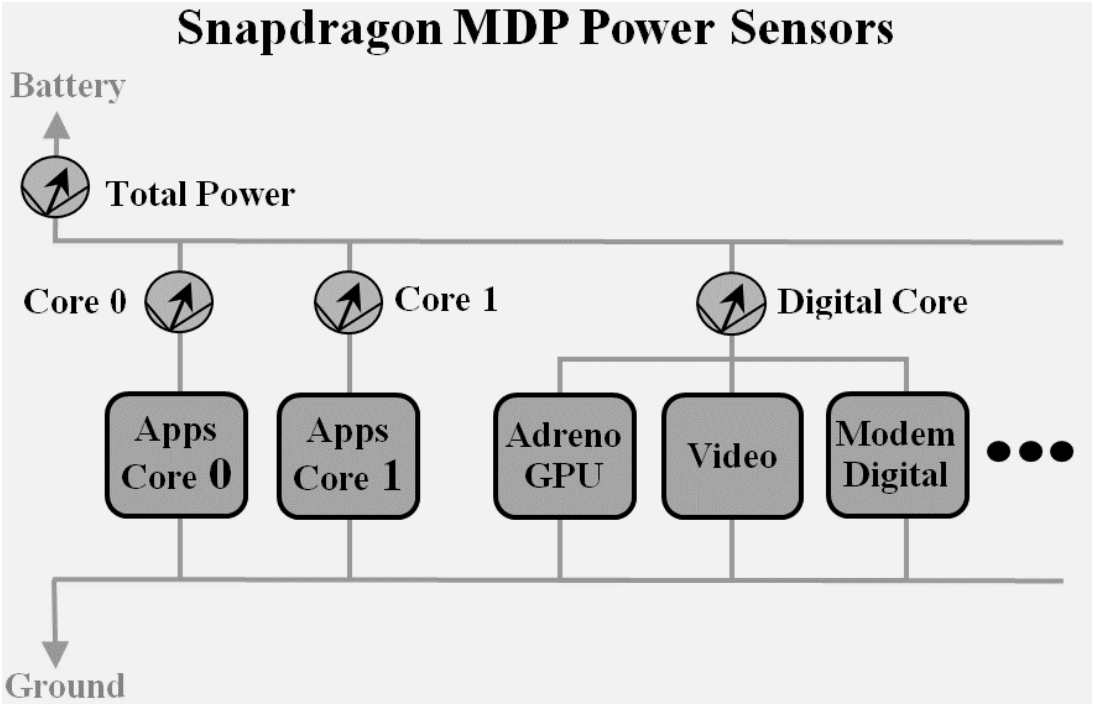


Figura 16. Funcionamiento del sensor de energía de los procesadores Snapdragon

Trepn Power Profiler permite recolectar los siguientes tipos de datos:

- Estadísticas de uso de CPU: Carga de CPU general, carga de CPU normalizada, y carga individuales de cada uno de los núcleos del CPU, hasta 8 núcleos.
  La carga de CPU representa el uso con respecto a la frecuencia de operación de CPU. La normalizada, representa el uso con respecto a la máxima frecuencia posible. Esto se debe a que cuando el procesador está ocioso, la frecuencia es reducida con el objetivo de ahorrar energía. Por ejemplo, si a un procesador con frecuencia máxima de 1000 MHz, se le reduce a 500 MHz, y el dispositivo requiere 100 MHz de procesamiento, la carga de CPU indicaría 20%, mientras que la carga de CPU normalizada indicaría 10%.
- Estadísticas de uso de GPU (siglas en inglés de Unidad de Procesamiento Gráfico). Frecuencia de GPU y porcentaje de uso de GPU.
- Uso de energía: mediciones de consumo de energía instantáneo, y promediado.
- Estado de las antenas: Estado del Bluetooth, datos móviles, Wi-Fi, Wi-Fi RSSI (por las siglas en inglés de Indicador de Fuerza de la Señal Recibida).
- Estadísticas de la pantalla: porcentaje de brillo de la pantalla, estado encendido o apagado.
- Uso de memoria RAM, porcentaje total y por aplicación.
- Otras estadísticas: uso de Wi-Fi y datos móviles por aplicación, estado de las aplicaciones, estado del GPS, Wakelocks, Wi-Fi locks (estos últimos dos son para dispositivos con Android 4.3 o inferior).

Esta información en conjunto permite comprender mejor el rendimiento energético para poder optimizar la aplicación monitoreada. Cuando se usa con los dispositivos Mobile Development Platform (MDP) basados en el procesador Snapdragon de Qualcomm, Trepn Profiler es capaz de mostrar el consumo de energía de los componentes de hardware individuales [69].

Trepn Profiler trabaja con tres niveles de granularidad: sistema, subcomponente y aplicación. En general, el *profiling* a nivel de subcomponente y de aplicación es más complejo que el *profiling* a nivel de sistema, simplemente porque se requiere información más detallada sobre el comportamiento del sistema subyacente y la ejecución de la aplicación.

Entre las características más interesantes de Trepn Profiler se incluye la capacidad de registrar estadísticas y visualizar información en gráficos en tiempo real. Además, las tablas con información y los gráficos superpuestos sobre la aplicación que se está monitoreando, permiten analizar dinámicamente en tiempo real el consumo de energía de dicha aplicación conforme el usuario interactúa con ella. Los datos también se pueden exportar para el análisis fuera de línea.

Debido a que el dispositivo Moto G2 utilizado en las pruebas de esta experimentación cuenta con un procesador Qualcomm Snapdragon 400, es posible realizar mediciones del consumo de energía de grano fino. Sin embargo, la granularidad de Trepn Profiler en relación con el consumo energético es por subcomponente y no por aplicación, ello obliga a tomar los recaudos del caso para minimizar las interferencias externas a la aplicación durante las mediciones. Para ello se ha establecido una serie de precondiciones que el dispositivo debe cumplir antes de comenzar la ejecución de una prueba.

Estas precondiciones son:

- El brillo de la pantalla debe estar configurado al nivel mínimo.
- El dispositivo debe estar configurado en modo avión.
- El nivel de carga de la batería del dispositivo debe estar entre 80% y 100%.
- El dispositivo no debe estar conectado al cargador al momento de realizar las pruebas.
- La aplicación de las pruebas debe tener fondo de color negro.
- La aplicación debe estar en primer plano en pantalla durante las pruebas.

Para realizar una medición confiable del consumo de energía de una aplicación es necesario considerar la naturaleza aleatoria de la experimentación. Un experimento aleatorio puede arrojar resultados diferentes bajo el mismo conjunto de condiciones iniciales. Es por esto que efectuar varias mediciones independientes del gasto energético de una aplicación, producirá un conjunto de resultados con ligeras diferencias entre sí. Por lo tanto, en el presente trabajo, para obtener un número X representativo del valor buscado, se calculó el promedio $\bar{X}$ de una serie de mediciones X_i efectuadas. En la mayoría de los casos de interés práctico 30 muestras resultan suficientes para postular a $\bar{X}$ como una buena aproximación a la media real de la distribución. Gran cantidad de trabajos experimentales publicados en el área utilizan este número de mediciones en su fase de recolección de datos, algunos de ellos han sido citados previamente en esta tesina [64] [1].

Para caracterizar cada una de las muestras obtenidas, se han calculado los estadísticos $\bar{X}$ y S_X que se corresponden con la media (o promedio muestral) y la desviación estándar muestral respectivamente, tal como se presentan en la Tabla 1.

Dada la muestra X = $X_1, X_2, \ldots, X_n$	
Media o promedio muestral	$\bar{X} = \left(\frac{1}{n}\right)\sum_{i=1}^{n} X_i$
Desviación estándar muestral	$S_X = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2}$

Tabla 1.Estadísticos utilizados en el análisis de los datos

Las variables estudiadas fueron consumo de energía, tiempo de ejecución y porcentaje de uso de CPU. Se calculó el promedio muestral y la desviación estándar muestral de la energía consumida en mWh (milivatios hora), el porcentaje de carga de la CPU y el tiempo de ejecución de las pruebas medido en segundos.

3.2.1.1 Aplicación con alto uso de CPU

Para simular una aplicación con alto uso del procesador, se implementó un algoritmo que realiza diversos cálculos aritméticos, calculando la siguiente serie matemática:

$$serie = \sum_{j=1}^{10} \sum_{k=1}^{1000000} \left(log_2(k) + \frac{3k}{2j} + \sqrt{k} + k^{j-1}\right)$$

El experimento planteado permite medir con precisión la variable analizada, en este caso, el gasto energético que implica el uso intensivo de CPU.

Como se mencionó anteriormente, el consumo de energía de la batería está fuertemente ligado al uso de CPU.

Este tipo de uso intensivo del CPU es frecuente en aplicaciones móviles que involucran procesamiento de imágenes, tales como juegos, edición de fotografías, realidad aumentada, entre otras, y donde no siempre es posible llevar el procesamiento a la GPU.

Este cálculo fue implementado para cada uno de los frameworks elegidos. El resultado fueron siete aplicaciones con idéntica funcionalidad.

Para este experimento se sumó el desarrollo de la aplicación usando el framework para Android NDK (Native Development Kit). [84] En este último se utilizó el lenguaje de programación C++. Este conjunto de herramientas, que se presenta como una opción dentro del enfoque nativo, permite incluir módulos implementados C y C++ en aplicaciones Android, con el objetivo de maximizar la performance, y reutilizar librerías implementadas en esos lenguajes.

El NDK suele ser usado en aplicaciones donde se necesite alcanzar un máximo rendimiento en el procesamiento de imágenes, generación de gráficos, simulaciones de física, etc. [85]

También es utilizado con el fin de portar código entre diferentes plataformas, ya que muchas de éstas permiten implementar módulos en C++.

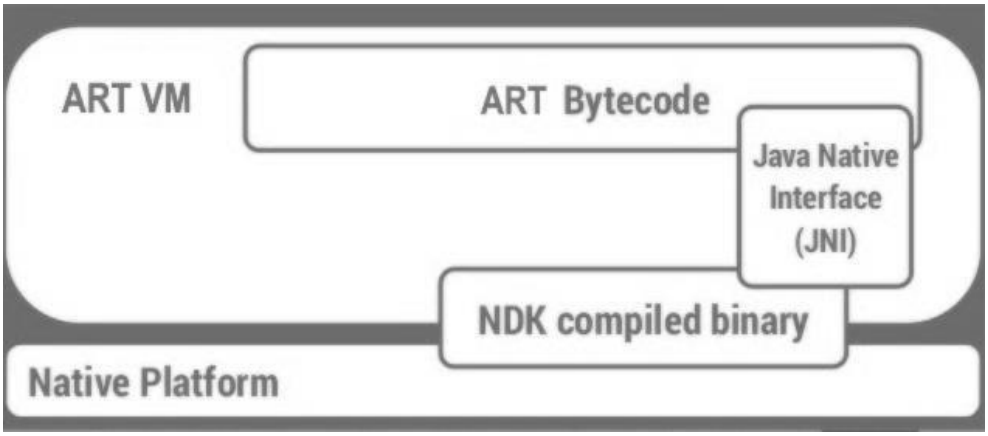


Figura 17. Ubicación de los binarios compilados con el NDK en la arquitectura de una aplicación Android

Los algoritmos implementados usando el NDK son compilados a ejecutables binarios específicos para cada una de las arquitecturas sobre las que Android se ejecuta. [86] El programador debe elegir cuáles arquitecturas quiere soportar. Esto hace que el código pase a ejecutarse directamente por el procesador, a diferencia del código Java, que es compilado a bytecode para ser ejecutado por la máquina virtual de Android, ART. En la figura 17 se observa la ubicación de los binarios compilados con el NDK.

Para cada uno de los 7 casos de prueba planteados, correspondientes a los frameworks de desarrollo ya mencionados, se realizaron 30 ejecuciones independientes registrando los datos de consumo de energía, tiempo de ejecución y porcentaje de uso de CPU.

El uso de energía es medido en unidades de Wh (Watt-hora), las cuales miden la cantidad de trabajo realizado durante un tiempo determinado.

A modo de ejemplo, en la figura 18 se incluye el código multiplataforma desarrollado en el lenguaje Lua para el framework Corona.

```lua
-----------------------------------------------------------------------------------------
--
-- main.lua
--
-----------------------------------------------------------------------------------------

local widget = require( "widget" )

local t = display.newText( "Waiting for button event...", 0, 0, native.systemFont, 18 )
t.x, t.y = display.contentCenterX, 70

local function onSystemEvent( event )
    if event.type == "applicationStart" then
        initial = os.clock()

        serie = 0;

        for j=1,10 do
            for k=1,1000000 do
                serie = serie + ( math.log(k) / math.log(2) ) + (3*k/2*j) + math.sqrt(k) + math.pow(k, j-1);
            end
        end

        final = os.clock()
        tiempo = final - initial;

        t.text = serie .. " -> " .. tiempo
    end

end
```

Figura 18. Código de la serie matemática, escrito en el lenguaje Lua

3.2.1.2 Aplicación de reproducción de video

En el siguiente experimento se buscó simular una aplicación con funcionalidades de multimedia con reproducción de video. Para éste, se desarrollaron seis aplicaciones, una con cada framework de desarrollo elegido, con la funcionalidad de reproducir un video a pantalla completa de un minuto de duración.

El video incluido fue un archivo de 89,2 megabytes, con resolución de alta definición estándar, es decir 1280 por 720 píxeles. Se usó el códec más común en el mercado, H264, con tasa de bits de 5585 kilobits por segundo [87].

Para cada uno de los 6 casos de prueba planteados, se realizaron 30 ejecuciones independientes registrando los datos de consumo de energía, tiempo de ejecución y porcentaje de uso de CPU.

3.2.1.3 Aplicación de reproducción de audio

En este experimento se buscó simular otro tipo muy común de aplicación, donde se reproducen archivos de audio. Para esto, se desarrollaron seis aplicaciones con la funcionalidad de reproducir un archivo de audio. Se eligió un archivo de un minuto de duración, con el códec estándar más popular en el mercado, MP3 AC3, con tasa de bits de 128 kilobits por segundo, y un tamaño de 1,32 megabytes.

Para cada uno de los 6 casos de prueba planteados, se realizaron 30 ejecuciones independientes registrando los datos de consumo de energía, tiempo de ejecución y porcentaje de uso de CPU.

3.2.2 Uso de espacio de almacenamiento

El objetivo de estos experimentos es evaluar qué impacto tiene el enfoque de desarrollo elegido sobre el tamaño final de las aplicaciones. Para esto, se midió el tamaño de los paquetes de aplicaciones generadas.

Los diferentes enfoques de desarrollo de aplicaciones multiplataforma implican distintas técnicas para la construcción de las aplicaciones y la ejecución del código, las cuales impactarán directamente en el tamaño final de la aplicación. Por ejemplo, una aplicación nativa será ejecutada de manera directa por el sistema operativo subyacente, lo que significa que no necesita empaquetar un intérprete dentro de la aplicación. Una aplicación híbrida desarrollada con herramientas web, puede estar diseñada para ejecutarse con el navegador web provisto por el sistema operativo, o puede incluir su propio navegador dentro de la aplicación, y esto repercutirá sobre el tamaño final.

Lo mismo ocurre con una aplicación interpretada, la cual tendrá una parte ejecutada directamente por el sistema operativo, y otra ejecutada por un intérprete, que puede ser interno o usar uno provisto por la plataforma.

La primera prueba consistió en aplicaciones idénticas con la funcionalidad básica de mostrar un texto en una pantalla blanca. A esta prueba, se le sumaron las aplicaciones creadas para las pruebas anteriores, de reproducción de audio y video. Al evaluar el uso de espacio de almacenamiento, resulta importante analizar diferentes funcionalidades, ya que dependiendo de la estrategia de desarrollo del framework, existe la posibilidad de que se incluyan librerías, módulos o plugins al momento de generar la aplicación. Por ejemplo, en el caso de NativeScript, no incluye ninguna herramienta para reproducir videos por defecto, sino que es responsabilidad del programador agregarla. Para estos experimentos, se usó la librería de reproducción de video sugerida por la documentación oficial. [88]

En todos los casos, se verificó que no se incluyeran archivos adicionales tales como imágenes o videos. Estos archivos suelen ser agregados por las herramientas de los frameworks al momento de crear una nueva aplicación, a modo de ejemplo de sus funcionalidades.

3.2.2.1 Aplicación basada en texto

El primer experimento consistió en generar aplicaciones simples que mostraran un texto en pantalla, y medir el tamaño de la aplicación generada por cada framework. El objetivo es conocer el tamaño mínimo que tendrá cada tipo de aplicación.

3.2.2.2 Aplicaciones de audio y video

En este caso, se analizará el tamaño de cada una de las aplicaciones generadas para los experimentos con aplicaciones de contenido multimedia. Según el enfoque utilizado, puede ocurrir que se incluyan diferentes librerías y herramientas, o que se dependa de la plataforma y no sea necesario incluir nada extra que aumente el tamaño de la aplicación.

Capítulo 4. Resultados obtenidos

4.1 Experimentación de uso de batería

En esta sección se reproducen los resultados de los experimentos desarrollados para evaluar el impacto de los diferentes enfoques de desarrollo en la batería de los dispositivos.

La figura 23 describe en detalle las mediciones del consumo energético obtenidas. Allí se grafican los histogramas que visualizan la distribución de las muestras recolectadas para cada uno de los 7 frameworks de desarrollo testeados.

4.1.1 Consumo de batería en aplicaciones con alto uso de CPU

La tabla 2 resume los resultados obtenidos durante las pruebas realizadas para analizar las aplicaciones Android con alta carga de procesamiento. Cada fila de datos se refiere a uno de los frameworks de desarrollo analizados, ordenados por consumo energético, comenzando por el más eficiente: Cordova, hasta el menos eficiente: Corona. La columna con encabezado "Energía" es la más relevante ya que presenta el promedio y desviación estándar muestrales del consumo de energía obtenidos durante la realización de las pruebas. Las otras columnas: "carga CPU" y "Duración", completan la descripción de los resultados y se refieren al porcentaje de CPU utilizado por la aplicación y al tiempo de ejecución que tomó la realización de la tarea respectivamente.

Framework	**Energía** (mWh.)		**Carga CPU** (%)		**Tiempo de ejecución** (s.)	
	$\overline{E}$	S_E	$\overline{C}$	S_C	$\overline{T}$	S_T
Cordova	1.597	0.136	35.924	2.571	8.467	0.679
Titanium	1.692	0.096	37.480	2.395	8.355	0.643
Android NDK	1.789	0.092	32.434	1.876	9.745	0.366
NativeScript	1.792	0.176	33.357	2.217	9.109	1.789
Xamarin	3.036	0.185	32.072	1.768	17.891	0.973
Android SDK	3.463	0.149	32.468	1.332	18.568	2.938
Corona	7.304	0.189	44.347	54.793	38.877	1.492

Tabla 2. Resultados de aplicación de procesamiento intensivo

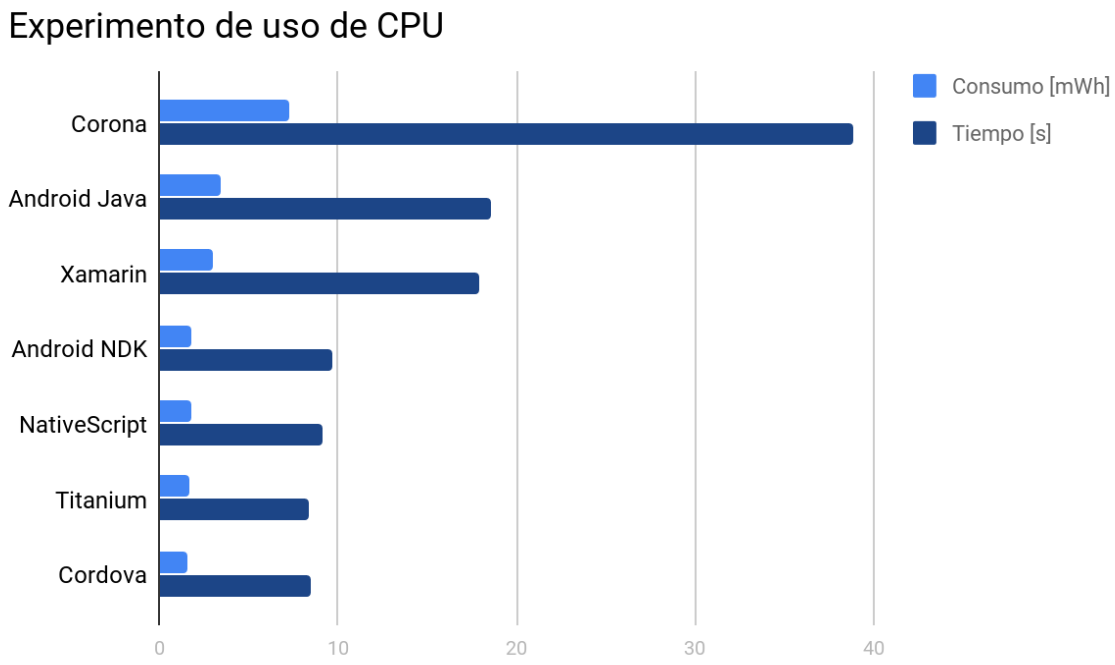


Figura 19. Resultados del experimento de procesamiento intensivo, ordenados por cantidad de energía utilizada

De estos resultados se desprende que aquellos frameworks que tuvieron mayor tiempo de ejecución, también tuvieron el mayor consumo de batería, sugiriendo un vínculo entre el uso del CPU y el consumo de energía.

En la figura 19 se presentan los resultados en un gráfico de barras, ordenados por el nivel de consumo energético.

Del análisis de estos datos en conjunto surgen con claridad 3 grupos de frameworks. El primero y de mayor eficiencia energética está integrado por Cordova, Titanium, Android nativo NDK y NativeScript. El segundo, de mediana eficiencia, lo integran Xamarin y Android nativo SDK. Por último, el grupo de menor eficiencia energética está conformado por Corona. Corona también fue el framework que produjo mayor carga de CPU y tiempo de ejecución (2 veces más que Android SDK (el más ineficiente excluyendo a Corona), a ello se debe su mayor consumo de energía.

Los frameworks con menor consumo de energía y menor tiempo de ejecución fueron Cordova, de enfoque híbrido, y NativeScript y Titanium, de enfoque interpretado. Lo que estos frameworks tienen en común, es el uso de código JavaScript.

Es interesante destacar la diferencia entre el enfoque nativo tradicional (Android con Java) y los frameworks que usan JavaScript (Cordova, NativeScript, Titanium). La versión nativa desarrollada en Java consume aproximadamente el doble que las versiones que utilizan JavaScript. Esto se debe principalmente al uso de las funciones matemáticas provistas en el lenguaje Java, que se ejecuta sobre ART, la máquina virtual de Android, y que muestran grandes deficiencias en relación al tiempo de ejecución y por

consiguiente también en relación al consumo de energía. Al ejecutarse en JavaScript, en cambio, se usará el intérprete de JavaScript del sistema operativo, en este caso Chrome V8. [89]

El enfoque nativo usando el NDK con C++, tuvo un consumo similar a los frameworks que usan JavaScript, por lo que se presenta como una buena opción al momento de implementar funcionalidades con mucha demanda de CPU, sin salir del entorno nativo.

Por último Xamarin tuvo un tiempo y consumo muy similar al de Android nativo con Java.

Si se multiplica $\bar{T} \times \bar{C}$ (tiempo empleado para realizar la tarea por el porcentaje de carga de CPU) y se ordenan los frameworks testeados por este valor en forma creciente se obtiene el siguiente resultado: 1) NativeScript, 2) Cordova, 3) Titanium, 4) Android NDK, 5) Xamarin, 6) Android SDK y 7) Corona.

Obsérvese que, salvo la posición de NativeScript, coincide con el ordenamiento según el consumo de energía que se muestra en la Tabla 2. Esto postula al producto del tiempo empleado por la carga de CPU como un buen estimador del consumo energético de la aplicación, lo que resulta conveniente dado que ambos parámetros son más sencillos de obtener que el consumo real de la aplicación.

4.1.2 Consumo de batería en aplicación de video

A las precondiciones mencionadas anteriormente, se sumó que el volumen del audio del dispositivo estuviera al 20%, para todas las pruebas.

Se tomaron mediciones de 30 ejecuciones para cada aplicación desarrollada, tomando los datos de consumo de energía, porcentaje de uso de CPU y tiempo de ejecución.

La tabla 3 resume los resultados obtenidos durante las pruebas de reproducción de video. Los frameworks de desarrollo analizados se muestran ordenados por consumo energético desde el más eficiente: Android SDK, hasta el menos eficiente: Cordova.

Framework	Energía (mWh.)		Carga CPU (%)		Tiempo de ejecución (s.)	
	$\bar{E}$	S_E	$\bar{C}$	S_C	$\bar{T}$	S_T
Android SDK	4.776	0.287	14.540	0.862	61.600	0.814
Corona	4.992	0.235	14.704	0.711	62.733	0.907
Xamarin	5.119	0.473	15.465	1.608	62.333	0.959
Titanium	5.262	0.502	15.204	1.643	63.633	1.033
NativeScript	11.112	1.590	17.839	2.210	63.333	1.295
Cordova	13.866	0.536	22.358	0.903	62.833	0.834

Tabla 3. Resultados de aplicación de reproducción de video

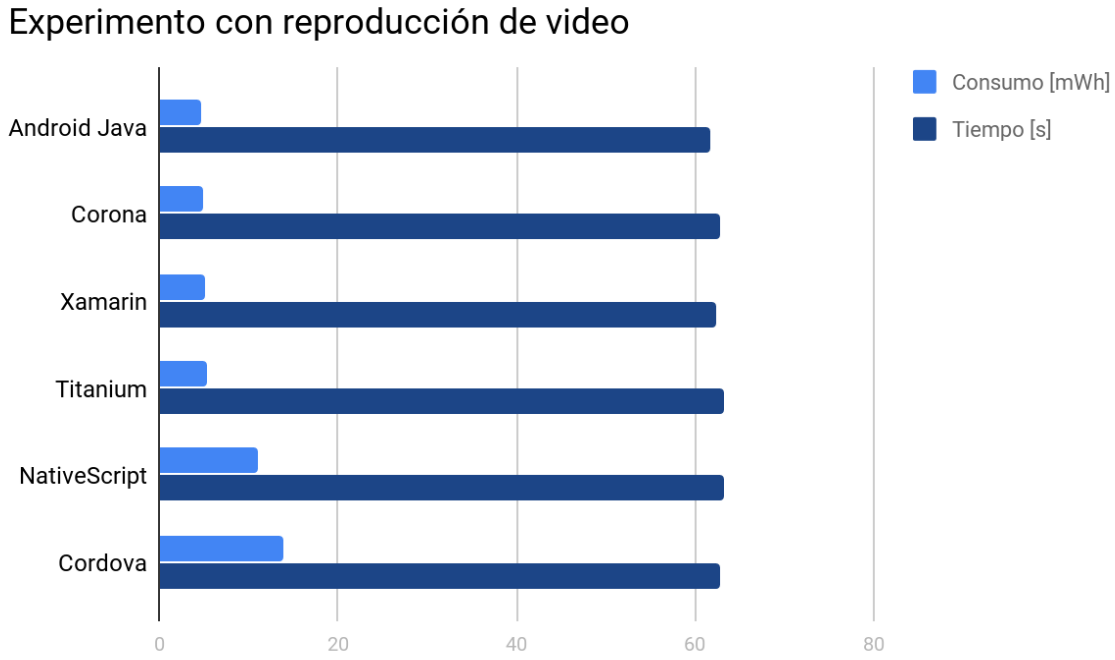


Figura 20. Resultados del experimento de reproducción de video, ordenados por cantidad de energía utilizada

Los resultados obtenidos son muy diferentes a los de la prueba que involucraba cálculos aritméticos.

Observando la figura 17 y la columna B de la figura 19, puede apreciarse 3 grupos, que coinciden con los enfoques de desarrollo. En primer lugar se encuentra el enfoque nativo, con el menor consumo, seguido por los frameworks con generación por compilación cruzada, y con un mayor consumo, los frameworks de enfoque interpretado. Por último el enfoque híbrido fue el peor en cuanto a eficiencia energética.

La aplicación con menor consumo fue la desarrollada con enfoque nativo en Java. Corona y Xamarin tuvieron un rendimiento muy cercano al nativo, seguido por NativeScript y Titanium. El peor enfoque en este caso fue el híbrido con Cordova, causando más del triple de consumo de energía que el enfoque nativo. Esto probablemente se debe al uso del reproductor de video HTML del navegador utilizado (en este caso, el *WebView* de Android basado en Chrome), el cual requiere más poder de cómputo que en el reproductor provisto por el sistema operativo.

Si bien la duración del video era de 60 segundos, se observa que el tiempo promedio varía ligeramente según el framework utilizado (ver Tabla 3). Esto se debe a que cada aplicación tiene diferentes tiempos de arranque, lo cual está directamente vinculado al enfoque de desarrollo del framework. El enfoque nativo muestra una clara ventaja en este aspecto, mientras que los demás muestran una demora similar al arrancar. De todos modos, esto no afecta en gran medida al consumo, ya que se trata de una diferencia de tiempo del 1 a 2% aproximadamente, y la mayor parte del consumo de CPU es realizado por el reproductor de video utilizado. Debe considerarse que esta diferencia es significativa sólo si el tiempo de uso de la aplicación es breve, perdiendo relevancia en caso contrario.

El ordenamiento de los frameworks por el valor del producto $\bar{T} \times \bar{C}$ es exactamente el mismo que el obtenido utilizando el valor de la energía consumida. Por lo tanto, al igual que en el caso de las aplicaciones con procesamiento intensivo, el producto del tiempo empleado por la carga de CPU es un buen estimador del consumo energético de la aplicación.

4.1.3 Consumo de batería en aplicación de audio

Al igual que en la prueba de aplicación de video, se cumplieron las precondiciones y se estableció el audio del dispositivo en 20%.

Se realizaron 30 ejecuciones del experimento, tomando los datos de consumo de energía, porcentaje de uso de CPU y tiempo de ejecución.

Los resultados se presentan en la tabla 4.

Framework	Energía (mWh.)		Carga CPU (%)		Tiempo de ejecución (s.)	
	$\overline{E}$	S_E	$\overline{C}$	S_C	$\overline{T}$	S_T
Android SDK	3.920	0.291	10.497	0.882	64.033	0.999
Xamarin	4.010	0.201	10.592	0.613	64.967	1.098
Titanium	4.189	0.277	11.865	0.835	64.767	1.104
NativeScript	4.224	0.229	11.233	0.644	65.867	1.042
Cordova	4.288	0.191	11.473	0.487	65.733	1.388
Corona	5.194	0.387	14.680	1.080	64.800	1.031

Tabla 4. Resultados de las mediciones en aplicación de reproducción de audio

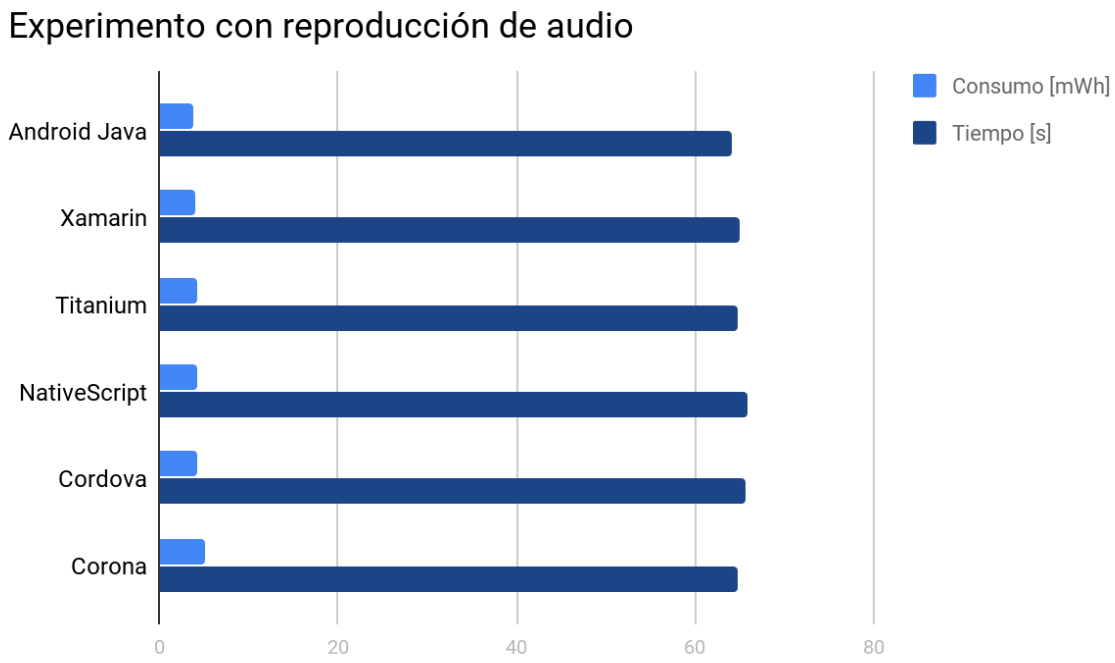


Figura 21. Resultados del experimento de reproducción de audio, ordenados por cantidad de energía utilizada

El análisis de la tabla 4 revela que no existen grandes diferencias en relación al consumo energético entre los distintos enfoques de desarrollo, a excepción de Corona que se distingue como el más ineficiente de todos, ubicándose con un 18% de consumo más que Cordova. Los histogramas de las muestras recogidas que se grafican en la figura 21 también apoyan esta conclusión. Se observa que en el caso de la reproducción de audio, la elección del framework de desarrollo tiene menor incidencia en comparación con los otros dos tipos de aplicaciones ya analizadas.

En general, se aprecia que la reproducción de audio resulta menos demandante en cuanto a procesamiento, comparado con la prueba de video.

Igualmente a lo ocurrido con la aplicación de reproducción de video, aquí también se observan diferencias en el tiempo de arranque de la aplicación de acuerdo al framework de desarrollo utilizado. El enfoque nativo nuevamente muestra una ligera ventaja en este aspecto.

El ordenamiento de los frameworks por el valor del producto $\bar{T} \times \bar{C}$ es muy similar al obtenido utilizando el valor de la energía consumida, coincidiendo en sus extremos y señalando la mayor diferencia entre Corona y el resto de los frameworks. Por lo tanto, en este caso también el producto del tiempo empleado por la carga de CPU es un buen estimador del consumo energético de la aplicación.

4.1.4 Análisis general

La figura 22 muestra los histogramas con la distribución de las muestras de consumo de energía recolectadas para cada uno de los 7 frameworks de desarrollo testeados.

En la figura 23 se grafican los valores de $\bar{E}$ obtenidos durante la ejecución de los 19 casos de prueba planteados. De una rápida inspección visual se desprende que en los tres tipos de aplicaciones analizados se destacan por ineficientes (alto consumo energético) sólo uno o dos frameworks de desarrollo. Tal es el caso de Corona para procesamiento intensivo, Cordova y NativeScript para reproducción de video y Corona para reproducción de audio. En el otro sentido, el framework más eficiente no se diferencia notablemente de otros frameworks también eficientes. Esto indica que conocer los frameworks a evitar en cada caso es de gran utilidad aun desconociendo cuál es la mejor opción de todas.

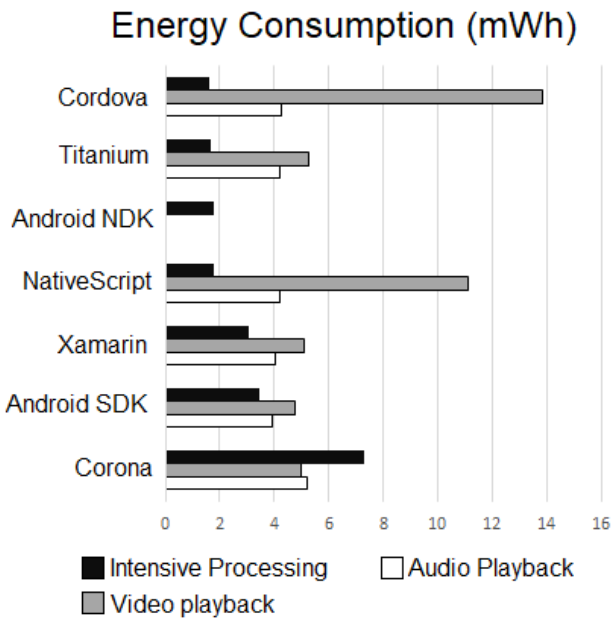


Figura 22.Consumo de energía según el framework de desarrollo utilizado para los tres tipos de aplicación estudiados

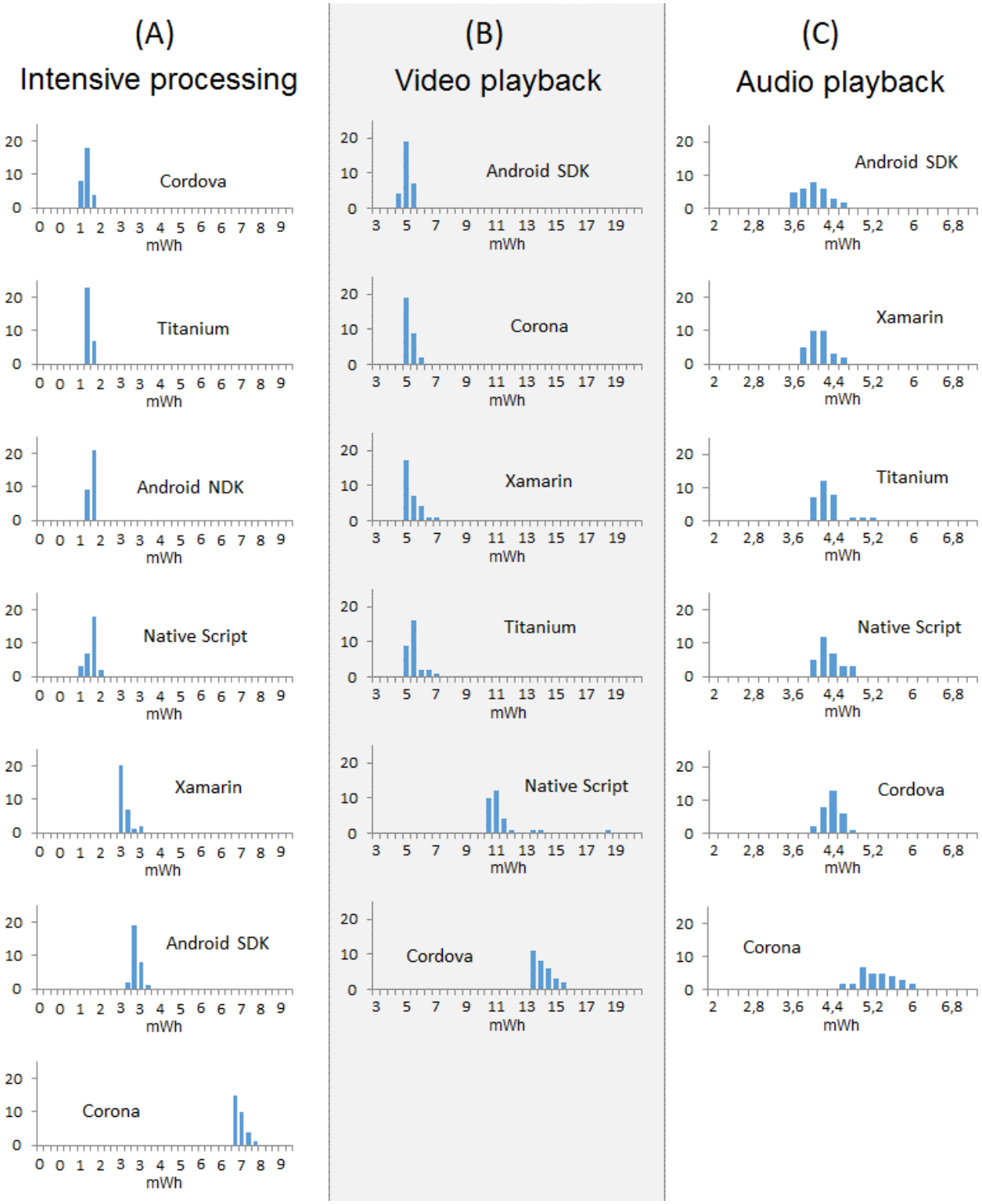


Figura 23. Histogramas sobre las muestras obtenidas durante la experimentación de uso de energía

De las tablas 2, 3 y 4 se evidencia que el impacto que tiene el framework de desarrollo utilizado sobre el consumo de energía, es mayor en las aplicaciones de procesamiento intensivo. Por ejemplo, el consumo de la aplicación desarrollada con Corona equivale a 4.57 veces el consumo de la aplicación desarrollada con Cordova.

El framework también resulta importante en las aplicaciones de reproducción de video. Por ejemplo, el consumo de la aplicación desarrollada con Cordova equivale a casi el triple del consumo de la aplicación desarrollada con Android SDK.

Finalmente, el menor impacto de la elección del framework de desarrollo se presenta en las aplicaciones de audio, en este caso el consumo del desarrollo con Corona equivale a 1.33 veces el consumo del desarrollo con Android SDK.

Se observa también que para estos tres tipos de aplicaciones estudiados, el framework de desarrollo Titanium, a pesar de que en ningún fue el mejor, siempre estuvo en el grupo de los más eficientes.

4.2 Experimentación de uso de espacio de almacenamiento

Para todas las pruebas, se generaron las aplicaciones de la forma estándar sugerida por la documentación de cada framework.

A continuación, se presentan los resultados de las mediciones en cuanto al uso de espacio de almacenamiento.

4.2.1 Uso de espacio de almacenamiento en aplicaciones basadas en texto

En la tabla 5 se presentan los tamaños de las aplicaciones desarrolladas.

Framework	Enfoque de desarrollo	Tamaño en megabytes
Android con Java	Nativo	1,48
Cordova	Híbrido	1,74
Corona	Generado por compilación cruzada	6,51
NativeScript	Interpretado	12,49
Titanium	Interpretado	8,54
Xamarin	Generado por compilación cruzada	4,08

Tabla 5. Tamaño de las aplicaciones basadas en texto

En este caso, la aplicación desarrollada siguiendo el enfoque nativo resultó la mejor con respecto al uso de espacio de almacenamiento. En segundo lugar estuvo Cordova, de enfoque híbrido, que produjo un paquete con solo 18% más tamaño que con el enfoque nativo.

Las aplicaciones generadas por compilación cruzada se ubicaron en un intermedio, Xamarin con 4,08 MB (176% más que la nativa) y Corona con 6,51MB (339% más que la nativa).

Por último, las aplicaciones creadas con los frameworks de enfoque interpretado tuvieron el mayor tamaño. Esto se debe a que tanto NativeScript como Titanium incorporan un intérprete de JavaScript dentro de la aplicación.

De estos datos se desprenden cuatro grupos de frameworks, coincidentes con el enfoque de desarrollo que utilizan. Los mejores resultaron el enfoque nativo y el híbrido. Esto se debe a que ambos enfoques prescinden de un intérprete, en el caso del enfoque nativo, usando las interfaces provistas por la plataforma, y en el caso del enfoque híbrido, usando código web que luego se ejecutará sobre el intérprete web que la plataforma provea. En segundo lugar en cuanto al uso de espacio estuvo el enfoque generado por compilación cruzada, y por último, el enfoque interpretado.

4.2.2 Uso de espacio de almacenamiento en aplicaciones de video

Para las aplicaciones con reproducción de video, también se utilizaron las técnicas sugeridas por la documentación de cada framework. El archivo de video añadido tenía un tamaño de 89,2 megabytes.

Como se mencionó anteriormente, la aplicación básica es un buen parámetro para conocer cuántos megabytes sumará cada framework. Sin embargo, es una buena idea probar otras funcionalidades como reproducción de archivos multimedia, ya que puede ocurrir que se incluyan otras herramientas o librerías para ese fin, que aumenten el tamaño de la aplicación.

En la tabla 6 se presentan los resultados.

Framework	Enfoque de desarrollo	Tamaño en megabytes	Tamaño restando el archivo de video
Android con Java	Nativo	90,24	1,04
Cordova	Híbrido	91,97	2,77
Corona	Generado por compilación cruzada	95,78	6,58
NativeScript	Interpretado	101,67	12,47
Titanium	Interpretado	98,43	9,23
Xamarin	Generado por compilación cruzada	94,21	5,01

Tabla 6. Tamaño de las aplicaciones de video

Para el caso de las aplicaciones de reproducción de video, los tamaños se mantuvieron en el orden observado en la aplicación de texto.

En la tercera columna de la tabla, se detalla el tamaño total de cada una de las aplicaciones generadas. En la cuarta columna, se detalla la diferencia entre el tamaño total y el tamaño del archivo de video, debido a que este archivo debe ser empaquetado junto con el código de cada aplicación. De esta forma, se puede diferenciar con más facilidad el tamaño que agrega cada framework, y también compararlo con el tamaño de la aplicación basada en texto.

En el caso del enfoque nativo, se mantuvo como la opción de menor tamaño, agregando solo un megabyte por sobre el tamaño del archivo de video. Esto es incluso menor que el tamaño resultante en la aplicación de texto.

En segundo lugar estuvo el enfoque híbrido, con Cordova, sumando 2,77 megabytes.

Luego siguieron los frameworks de enfoque generado por compilación cruzada, donde Corona sumó 6,58 megabytes y Xamarin 5,01 megabytes.

Por último, los peores resultados fueron arrojados por los frameworks de enfoque interpretado. Al igual que se observó en la aplicación basada en texto, incorporar un intérprete dentro de la aplicación hace que NativeScript y Titanium generen las aplicaciones más pesadas, sumando aproximadamente 12 y 9 megabytes respectivamente.

4.2.3 Uso de espacio de almacenamiento en aplicaciones de audio

Las aplicaciones con reproducción de audio fueron generadas de la forma estándar sugerida por cada framework, y además se utilizó la funcionalidad de reproducción de audio que se mencionara en la documentación de cada framework. El archivo de audio utilizado tiene un tamaño de 1,32 megabytes.

En la tabla 7 se presentan los resultados de la aplicación de audio. Se mantiene la agrupación por enfoque de desarrollo observada anteriormente.

Framework	Enfoque de desarrollo	Tamaño en megabytes	Tamaño restando el archivo de audio
Android con Java	Nativo	2,7	1,38
Cordova	Híbrido	3,14	1,82
Corona	Generado por compilación cruzada	8,05	6,73
NativeScript	Interpretado	22,43	21,11
Titanium	Interpretado	10,48	9,16
Xamarin	Generado por compilación cruzada	5,4	4,08

Tabla 7. Tamaño de las aplicaciones de audio

En primer lugar, las aplicaciones nativa e híbrida resultaron las mejores. Luego estuvieron las aplicaciones con enfoque generado, y por último las de enfoque interpretado.

4.2.4 Análisis general

En la figura 24, se grafican los resultados de las mediciones de las tres aplicaciones, ordenados por el tamaño final de las aplicaciones. Inmediatamente se visualiza que el aumento en el tamaño de las aplicaciones fue lineal dentro de cada framework. Es decir, en los tres tipos de aplicaciones, el tamaño final según el framework se mantuvo en el mismo orden.

La diferencia entre el mejor y el peor framework resulta porcentualmente importante en las aplicaciones de texto y de audio, no tanto así en las aplicaciones de video. Esto se debe a que el mayor impacto del framework en el tamaño de la aplicación se produce al realizar el empaquetamiento de los componentes básicos de la aplicación. Es decir, las librerías, intérpretes y demás recursos que el framework necesita incluir dentro de la aplicación para funcionar.

De esta forma, las aplicaciones creadas usando los enfoques nativo e híbrido resultaron las de tamaño más reducido, ya que en éstos no se incluye un intérprete dentro de la aplicación.

En segundo lugar están los frameworks de enfoque de compilación cruzada, y por último los frameworks de enfoque interpretado.

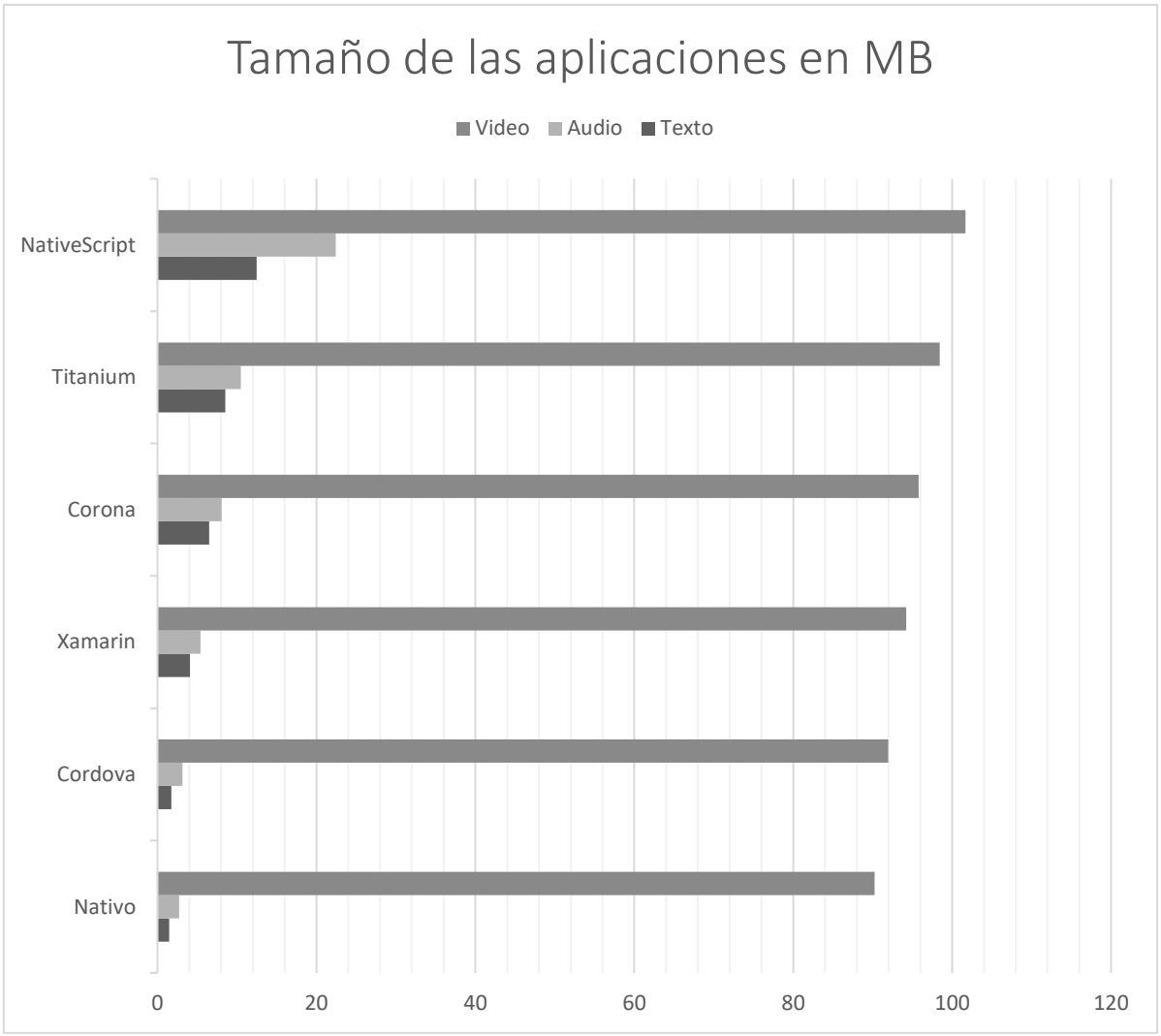


Figura 24. Resumen de las mediciones de espacio de almacenamiento

Capítulo 5. Conclusiones

Originalmente los dispositivos móviles, más específicamente los teléfonos móviles, fueron pensados con objetivos muy específicos. Con el avance de la tecnología móvil, las telecomunicaciones y el aumento de poder de cómputo, se generaron nuevas posibilidades de uso, transformándose en teléfonos inteligentes de uso masivo.

Sin embargo, el área de desarrollo de aplicaciones móviles tiene particularidades que no tenían tanta presencia en los desarrollos de software tradicionales, tales como la necesidad de utilizar baterías, conexiones a internet inestables, tiempos de desarrollo acotados, y una gran fragmentación entre las diferentes plataformas.

Todo esto trajo nuevos requerimientos a los que la ingeniería de software debe adaptarse.

Es por esto que surgen diferentes enfoques de desarrollo de aplicaciones móviles. En primer lugar, el enfoque nativo, que implica utilizar las herramientas provistas por los fabricantes de cada una de las plataformas a las que se quiera llegar. Esto trae las desventajas de tener que encarar un desarrollo por separado para cada plataforma. Apuntando a mejorar esta situación, es que surgen los frameworks multiplataforma. Estos a su vez tienen diferentes enfoques para la generación de aplicaciones en base a un único desarrollo.

En este trabajo, se analizaron los enfoques de desarrollo nativo, interpretado, híbrido y generado por compilación cruzada.

Para realizar un análisis de los enfoques de desarrollo, que pueda ser de utilidad para un ingeniero de software al momento de tomar la decisión de cuál enfoque usar en un nuevo proyecto, se diseñaron una serie de casos de prueba. Para ello, se estudiaron las principales problemáticas del desarrollo de aplicaciones móviles, y se eligieron dos factores que pueden resultar claves para el éxito de una aplicación, ya que impactan directamente en recursos que son muy limitados en el contexto de la computación móvil: el consumo de energía de la batería, y el uso de espacio de almacenamiento persistente.

Para evaluar el uso de energía, se desarrollaron tres tipos de aplicaciones, cada una de ellas implementada en todos los enfoques de desarrollo estudiados. La primera fue una aplicación que realiza procesamiento intensivo, la segunda, una aplicación de reproducción de video, y por último una aplicación de reproducción de audio. Como resultado de esto, se desarrollaron 19 aplicaciones experimentales, que se utilizaron para realizar mediciones de consumo de energía, utilizando una herramienta de software especializada para esto.

Los resultados arrojados por estas mediciones, si bien no permiten destacar a un único framework como el mejor, posibilitan un análisis comparativo interesante.

Se determinó que Titanium (perteneciente al enfoque de desarrollo multiplataforma interpretado) es el único framework de desarrollo con el que se consiguen resultados energéticamente eficientes para los tres tipos de aplicaciones testeados. Por lo tanto, en cuanto a consumo de energía, su utilización resulta apropiada en todos los casos alcanzados por este estudio.

El desempeño de otros frameworks de desarrollo fue muy variante en función del tipo de aplicación implementada, por ejemplo Cordova mostró la mayor eficiencia energética en una aplicación de procesamiento intensivo pero la peor en una aplicación de reproducción de video.

El análisis de los datos sugiere que descubrir cuál es el framework de desarrollo más eficiente para utilizar, no es tan importante como determinar cuáles son los menos eficientes para evitarlos. Esto se debe a que en los tres escenarios testeados la diferencia entre el mejor y el resto no resultó significativa; sin embargo se encontraron grandes diferencias entre el peor (o peores) y el resto de los frameworks.

Las consecuencias de una mala elección en el framework de desarrollo se maximizan con las aplicaciones que realizan procesamiento intensivo. En este escenario el desarrollo con Corona prácticamente quintuplica el consumo energético que se obtiene al desarrollar con Cordova. Otro tipo de aplicaciones muy sensible al framework de desarrollo es el de reproducción de video, en este caso una aplicación desarrollada con Cordova consume casi el triple de energía que una aplicación Android nativa desarrollada con el SDK. Finalmente debe indicarse que en las aplicaciones de reproducción de audio, los resultados de consumo de energía fueron similares.

Como resultado colateral de este experimento, se señala que el producto del tiempo de ejecución por la carga de CPU constituye un buen indicio del consumo energético de una aplicación. El valor de este producto puede utilizarse con fines comparativos, lo que resulta conveniente debido a que la carga de CPU y el tiempo de ejecución son más fáciles de obtener que los valores reales de consumo.

Para evaluar el uso de espacio de almacenamiento, se tuvieron en cuenta tres tipos de aplicaciones. Por un lado, se utilizaron las aplicaciones de reproducción de audio y de video mencionadas anteriormente. Además, se generaron aplicaciones con la única funcionalidad de mostrar un texto en pantalla, con el objetivo de conocer el tamaño base de una aplicación.

A partir de este experimento, se pudieron extraer conclusiones sobre el uso del almacenamiento que hace cada framework, que resultaron coincidentes con los enfoques de desarrollo.

El impacto que el enfoque de desarrollo tendrá sobre el tamaño final de la aplicación es especialmente importante en aplicaciones con un conjunto de funcionalidades reducido, ya que en estos casos, será mucho más evidente el cambio en el tamaño final. Por ejemplo, en la aplicación basada en texto, la

desarrollada usando NativeScript resultó 8 veces más grande que la desarrollada usando las herramientas nativas.

Debido a que esta diferencia de tamaño es causada por la inclusión de librerías, intérpretes y otras herramientas de cada framework, este aumento no es lineal, sino que representa una diferencia de tamaño inicial, que según las funcionalidades implementadas en la aplicación, puede seguir creciendo, pero no en forma lineal. Por ejemplo, para el caso de la aplicación de video, la aplicación desarrollada con NativeScript es 1,12 veces más grande que la aplicación nativa.

Se determinó que el enfoque nativo, y el híbrido, resultan ser los mejores en cuanto a uso del almacenamiento. En un segundo lugar, se ubican los frameworks de enfoque generado por compilación cruzada, el cual se presenta como una alternativa razonable, ya que el tamaño que agrega a la aplicación es relativamente poco, especialmente a medida que la aplicación crece.

Luego se ubicaron los frameworks de enfoque interpretado, con las aplicaciones de mayor tamaño. Esto se debe principalmente a que en este enfoque de desarrollo, una parte del código se traduce a código nativo, y el resto es interpretado en tiempo de ejecución, y para esto se incluye un intérprete dentro del paquete de la aplicación.

El análisis integral de los datos permite elaborar algunas conclusiones.

De los resultados de la experimentación de consumo de batería, si bien no se pudo establecer un framework específico como el mejor, y se concluyó que la eficiencia es muy dependiente del tipo de aplicación a desarrollar, Titanium se presentó como una opción balanceada en todos los casos. Para el caso de las aplicaciones multimedia, el enfoque nativo resultó el mejor, y para las aplicaciones con procesamiento intensivo, los frameworks que utilizan JavaScript tuvieron una ventaja, seguidos del enfoque nativo usando C++.

De los resultados de las mediciones de uso de espacio de almacenamiento, se observó una agrupación según el enfoque de desarrollo. Los enfoques nativo e híbrido resultaron ser los más eficientes. La diferencia con los otros frameworks se achica a medida que el tamaño de la aplicación crece, ya que este tamaño inicial viene atado a las herramientas que el framework necesita incluir en el paquete de la aplicación.

De este modo, el enfoque interpretado queda en desventaja, lo cual debe ser tenido en cuenta para aplicaciones de funcionalidades reducidas, para las cuales resulta fundamental mantener al mínimo el tamaño final en megabytes si se busca una buena recepción en el mercado [13]. A medida que se incluyen más recursos y funcionalidades, y el tamaño de la aplicación crece, la diferencia entre este enfoque de desarrollo y los anteriores se hará cada vez menos notoria.

Es necesario considerar que los resultados aquí presentados están ligados al estado de arte de los frameworks de desarrollo examinados en el momento de

la escritura de este artículo y, por lo tanto, podrían variar en el futuro conforme estos frameworks vayan evolucionando.

5.1 Trabajo futuro

El estudio presentado en este trabajo se centra en el análisis del impacto que tienen los distintos enfoques y frameworks de desarrollo sobre el consumo energético de las aplicaciones Android. La elección de la plataforma objetivo se debió a que Android es el sistema operativo con mayor presencia en el mercado de los dispositivos móviles. Sin embargo, es importante también hacer foco sobre otras plataformas como iOS y Windows Phone, que se deja planteado como trabajo futuro.

Además, se estudiaron diversas herramientas de desarrollo de aplicaciones móviles. Sin embargo, debido al incesante avance de este campo, han surgido nuevas herramientas que van ganando terreno en el mercado. Se plantea como trabajo futuro el estudio de algunos de estos frameworks, tales como React Native [90], Flutter [91], Ionic [4], Kivy [92], Unity [93], entre otros.

Por otro lado, también se plantea la continuidad del análisis comparativo, agregando otros parámetros de comparación que resultan de gran importancia, tales como la velocidad de acceso al almacenamiento de datos, el tiempo de carga de las interfaces visuales y el tiempo de respuesta ante las interacciones iniciadas por el usuario.

Bibliografía

[1] C. Siebra, «The software perspective for energy-efficient mobile applications development,» de *Proceedings of the 10th International Conference on Advances in Mobile Computing & Multimedia*, 2012.

[2] «Stat Counter. Mobile & Tablet Operating System Market Share Worldwide Jan 2013 - July 2017,» [En línea]. Available: http://gs.statcounter.com/os-market-share/mobile-tablet/worldwide/#monthly-201301-201707.

[3] «Apache Cordova,» [En línea]. Available: https://cordova.apache.org/. [Último acceso: Noviembre 2017].

[4] «Ionic Framework,» [En línea]. Available: https://ionicframework.com/. [Último acceso: Noviembre 2017].

[5] «Appcelerator Titaniu,,» [En línea]. Available: https://www.appcelerator.org/. [Último acceso: Noviembre 2017].

[6] «NativeScript,» [En línea]. Available: https://www.nativescript.org/. [Último acceso: Noviembre 2017].

[7] S. Xanthopoulos y S. Xinogalos, «A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications,» de *Proceedings of the 6th Balkan Conference in Informatics*, Thessaloniki, Grecia, 2013.

[8] R. Santiago, S. Trabaldo, M. Kamijo y Á. Fernández, Mobile Learning: Nuevas realidades en el aula, Editorial Óceano.

[9] «Smart Insights. Statistics on consumer mobile usage and adoption,» [En línea]. Available: https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/. [Último acceso: Noviembre 2017].

[10] «Android Authority. People are using more apps now than ever before,» [En línea]. Available: http://www.androidauthority.com/people-are-using-apps-more-than-ever-before-769946/. [Último acceso: Noviembre 2017].

[11] «Qualcomm Snapdragon 835,» [En línea]. Available: https://www.qualcomm.com/products/snapdragon/processors/835. [Último acceso: Noviembre 2017].

[12] K. Vandenbroucke, D. Ferreira, J. Goncalves, V. Kostakos y K. D. Moor, «Mobile cloud storage: a contextual experience,» *Proceedings of the 16th international conference on Human-computer interaction with mobile devices & services (MobileHCI '14),* pp. 101-110, 2014.

[13] S. Tolomei, «Shrinking APKs, growing installs,» 20 Noviembre 2017. [En línea]. Available: https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2. [Último acceso: Noviembre 2017].

[14] J. Courville y F. Chen, «Understanding storage I/O behaviors of mobile applications,» de *32nd Symposium on Mass Storage Systems and Technologies (MSST)*, Santa Clara, CA, 2016.

[15] B. Flipsen, J. Geraedts, A. Reinders, C. Bakker, I. Dafnomilis y A. Gudadhe, «Environmental sizing of smartphone batteries,» de *Electronics Goes Green 2012+*, Berlin, 2012.

[16] A. Banerjee y A. Roychoudhury, «Future of mobile software for smartphones and drones: Energy and performance,» de *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, 2017.

[17] «Business Insider. Yes, it's safe to leave your smartphone plugged into the charger overnight.,» [En línea]. Available: http://www.businessinsider.com/safe-charge-smartphone-overnight-2017-8. [Último acceso: Noviembre 2017].

[18] C. Siebra, «Towards a green mobile development and certification,» de *Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2011.

[19] X. Chen, Y. Chen, Z. Ma y F. C. A. Fernandes, «How is energy consumed in smartphone display applications?.,» de *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications (HotMobile '13)*, New York, NY, USA, 2013.

[20] «ARM, big.LITTLE technology,» [En línea]. Available: https://developer.arm.com/technologies/big-little. [Último acceso: Noviembre 2017].

[21] A. K. s. Wong, T. K. Woo, A. T. L. Lee, X. Xiao, V. W. H. Luk y K. W. Cheng, «An AGPS-based elderly tracking system,» de *Ubiquitous and Future Networks, 2009*, Hong Kong, 2009.

[22] B. H. Matthias Böhmer, J. Schöning, A. Krüger y G. Bauer, «Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage,» de *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services (MobileHCI '11)*, New York, 2011.

[23] D. E. Conroy, C.-H. Yang y J. P. Maher, «Behavior Change Techniques in Top-Ranked Mobile Apps for Physical Activity,» *In American Journal of Preventive Medicine,* vol. 46, nº 6, pp. 649-652, 2014.

[24] C. L. Ventola, «Mobile Devices and Apps for Health Care Professionals: Uses and Benefits,» de *Pharmacy and Therapeutics 39.5*, 2014.

[25] «Stat Counter. Mobile & Tablet Operating System Market Share Argentina,» [En línea]. Available: http://gs.statcounter.com/os-market-share/mobile-tablet/argentina/#monthly-201301-201707. [Último acceso: Noviembre 2017].

[26] «Statista. Number of apps in leading app stores 2017,» [En línea]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. [Último acceso: Noviembre 2017].

[27] «Statista. Number of available applications in the Google Play Store from

December 2009 to September 2017,» [En línea]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. [Último acceso: Noviembre 2017].

[28] «Documentación de Android. Kotlin and Android,» [En línea]. Available: https://developer.android.com/kotlin/index.html. [Último acceso: Noviembre 2017].

[29] «Android Developers Blog. Android Announces Support for Kotlin,» [En línea]. Available: https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html. [Último acceso: Noviembre 2017].

[30] «Apple App Store,» [En línea]. Available: https://developer.apple.com/support/app-store/. [Último acceso: Noviembre 2017].

[31] «Microsoft Visual Studio,» [En línea]. Available: https://www.visualstudio.com/es/downloads/. [Último acceso: Noviembre 2017].

[32] «Bootstrap,» [En línea]. Available: http://getbootstrap.com/. [Último acceso: Noviembre 2017].

[33] «Google Developers. Web Push Notifications: Timely, Relevant, and Precise,» [En línea]. Available: https://developers.google.com/web/fundamentals/push-notifications/?hl=es. [Último acceso: Noviembre 2017].

[34] «Mozilla Web Docs. Web Workers,» [En línea]. Available: https://developer.mozilla.org/es/docs/Web/API/Worker. [Último acceso: Noviembre 2017].

[35] «Google Developers. Progressive Web Apps,» [En línea]. Available: https://developers.google.com/web/progressive-web-apps/. [Último acceso: Noviembre 2017].

[36] «Crosswalk Project,» [En línea]. Available: https://crosswalk-project.org/. [Último acceso: Noviembre 2017].

[37] «CocoonJS,» [En línea]. Available: https://ludei.com/cocoonjs. [Último acceso: Noviembre 2017].

[38] «jQuery,» [En línea]. Available: https://jquery.com/. [Último acceso: Noviembre 2017].

[39] «Materialize CSS,» [En línea]. Available: http://materializecss.com/. [Último acceso: Noviembre 2017].

[40] «Angular JS,» [En línea]. Available: https://angularjs.org/. [Último acceso: Noviembre 2017].

[41] «Ionic Framework. How 2015 went for Ionic,» [En línea]. Available: http://blog.ionic.io/how-2015-went-for-ionic/. [Último acceso: Noviembre 2017].

[42] «TypeScript,» [En línea]. Available: http://www.typescriptlang.org/. [Último acceso: Noviembre 2017].

[43] «Sencha ExtJS,» [En línea]. Available: https://www.sencha.com/products/extjs/#overview. [Último acceso: Noviembre 2017].

[44] «Sencha Touch,» [En línea]. Available: https://www.sencha.com/products/touch/. [Último acceso: Noviembre 2017].

[45] «NativeScript. Frequently asked questions.,» [En línea]. Available: https://www.telerik.com/platform/nativescript/faq. [Último acceso: Noviembre 2017].

[46] «Telerik. NativeScript – a Technical Overview,» [En línea]. Available: https://developer.telerik.com/featured/nativescript-a-technical-overview/. [Último acceso: Noviembre 2017].

[47] «Appcelerator Hyperloop,» [En línea]. Available: http://www.appcelerator.com/blog/2016/08/hyperloop-is-here/. [Último acceso: Noviembre 2017].

[48] J. v. Brocke, R. Hekkala, S. Ram y M. Rossi, «Design Science at the Intersection of Physical and Virtual Design,» de *8th International Conference, DESRIST 2013*, Helsinki, 2013.

[49] «Applause,» [En línea]. Available: https://github.com/applause/applause/tree/applause2. [Último acceso: Noviembre 2017].

[50] «iPhonical,» [En línea]. Available: https://code.google.com/archive/p/iphonical/. [Último acceso: Noviembre 2017].

[51] F. Zammetti, Learn Game Development with Corona SDK, Primera ed., Apress, 2013.

[52] «Corona FAQs,» [En línea]. Available: https://coronalabs.com/faq/. [Último acceso: Noviembre 2017].

[53] «Xamarin,» [En línea]. Available: https://www.xamarin.com/platform. [Último acceso: Noviembre 2017].

[54] «Xamarin. Understanding the Xamarin Mobile Platform,» [En línea]. Available: https://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_1_-_understanding_the_xamarin_mobile_platform/. [Último acceso: Noviembre 2017].

[55] A. Mesbah, P. Kruchten y M. E. Joorabchi, «Real Challenges in Mobile App Development,» de *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, 2013.

[56] A. Nitze, A. Schmietendorf y F. Rösler, «Towards a Mobile Application Performance Benchmark,» de *ICIW 2014: The Ninth International Conference on Internet and Web Applications and Services*, Paris, 2014.

[57] L. Delía, N. Galdamez, P. Thomas, L. Corbalán y P. Pesado, «Approaches to

Mobile Application Development: Comparative Performance Analysis,» de *2017 Computing Conference*, Londres, 2017.

[58] Qualcomm, «When mobile apps use too much power: A Developer Guide for Android App Performance. Documento técnico de Qualcomm,» [En línea].

[59] Adweek, «Facebook Apps Draining Your Batteries? Try Uninstalling And Using Its Mobile Site,» [En línea]. Available: http://www.adweek.com/digital/uinstall-facebook-apps-use-mobile-site/?red=af. [Último acceso: Noviembre 2017].

[60] N. Vallina-Rodriguez y J. Crowcroft, «Energy Management Techniques in Modern Mobile Handsets,» *IEEE Communications Surveys & Tutorials,* vol. 15, nº 1, pp. 179-198, 2013.

[61] H. Hassan y A. S. Moussa, «Power Aware Computing Survey,» *International Journal of Computer Applications,* vol. 90, nº 3, 2014.

[62] M. A. Hoque, M. Siekkinen y J. K. Nurminen, «Energy Efficient Multimedia Streaming to Mobile Devices — A Survey,» *IEEE Communications Surveys & Tutorials,* vol. 16, nº 1, pp. 579-597, 2014.

[63] P. Larsson, «Energy-efficient software guidelines,» Intel Software Solutions Group, Tech. Rep, 2011.

[64] L. Cruz y R. Abreu, «Performance-based guidelines for energy efficient mobile applications,» de *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, 2017.

[65] A. Pathak, Y. C. Hu y a. M. Zhang, « Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices,» de *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, Nueva York, 2011.

[66] Monsoon Power Monitor, [En línea]. Available: https://www.msoon.com/product-page/hv-power-monitor. [Último acceso: Noviembre 2017].

[67] M. A. Hoque, M. Siekkinen, K. N. Khan, Y. Xiao y S. Tarkoma, «Modeling, Profiling, and Debugging the Energy Consumption of Mobile Devices,» *ACM Computing Surveys ,* vol. 48, nº 3, 2016.

[68] Y. C. H. M. Z. Abhinav Pathak, «Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof,» de *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys '12)*, Nueva York, 2012.

[69] Qualcomm, «Trepn Profiler Starter Edition. User Guide,» Documento técnico de Qualcomm, 2015.

[70] W. S. a. M. B. Grace Metri, «Energy-Efficiency Comparison of Mobile Platforms and Applications: A Quantitative Approach,» de *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications (HotMobile '15)*, Nueva York, 2015.

[71] X. Zhang, A. Kunjithapatham, S. Jeong y S. Gibbs, «Towards an Elastic

Application Model for Augmenting the Computing Capabilities of Mobile Devices with Cloud Computing,» *Mobile Networks and Applications,* vol. 16, nº 3, p. 270–284, 2011.

[72] J. H. Christensen, «Using RESTful web-services and cloud computing to create next generation mobile applications,» de *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, Nueva York, 2009.

[73] Y. Lyu, J. Gui, M. Wan y W. G. J. Halfond, «An Empirical Study of Local Database Usage in Android Applications,» de *IEEE International Conference on Software Maintenance and Evolution* , Shanghai, China, 2017.

[74] G. Z. David T. Nguyen, X. Qi, G. Peng, J. Zhao, T. Nguyen y D. Le, «Storage-aware smartphone energy savings,» de *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, Nueva York, 2013.

[75] H. Kim, N. Agrawal y C. Ungureanu, «Revisiting storage for smartphones,» *ACM Transactions on Storage,* vol. 8, nº 4, 2012.

[76] G. Atanasov, «NativeScript Android Application Package Size Revealed,» [En línea]. Available: https://www.nativescript.org/blog/nativescript-android-application-package-size-revealed.

[77] «Google Play Store Multiple APK Support,» [En línea]. Available: https://developer.android.com/google/play/publishing/multiple-apks.html. [Último acceso: Noviembre 2017].

[78] «Optimising Titanium App File Sizes,» [En línea]. Available: https://www.appcelerator.com/blog/2017/07/optimising-titanium-app-file-sizes/. [Último acceso: Noviembre 2017].

[79] «APK Expansion Files,» [En línea]. Available: https://developer.android.com/google/play/expansion-files.html. [Último acceso: Noviembre 2017].

[80] K. K. a. Y. H. Lu, «Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?r,» *Computer,* vol. 43, nº 4, pp. 51-56, 2010.

[81] «Crosswalk Frequently-asked questions,» [En línea]. Available: https://crosswalk-project.org/documentation/about/faq.html. [Último acceso: Noviembre 2017].

[82] «Repositorio Git de los experimentos realizados,» [En línea]. Available: https://gitlab.com/cuiti/experimentos-enfoques-aplicaciones.

[83] «Trepn Profiler en Google Play Store,» [En línea]. Available: https://play.google.com/store/apps/details?id=com.quicinc.trepn. [Último acceso: Noviembre 2017].

[84] «Android NDK,» [En línea]. Available: https://developer.android.com/ndk/index.html. [Último acceso: Noviembre 2017].

[85] A. NDK, «Getting Started with the NDK,» [En línea]. Available: https://developer.android.com/ndk/guides. [Último acceso: Noviembre 2017].

[86] A. NDK, « CPUs and Architectures,» [En línea]. Available: https://developer.android.com/ndk/guides/arch.html. [Último acceso: Noviembre 2017].

[87] Blackbox, «hat is the H.264 video coding format and why it's becoming industry standard for video compression,» [En línea]. Available: https://www.blackbox.nl/en-nl/page/38301/Resources/Technical-Resources/Black-Box-Explains/multimedia/H264-video-compression. [Último acceso: Noviembre 2017].

[88] B. Martin, «NativeScript Videoplayer,» [En línea]. Available: https://github.com/bradmartin/nativescript-videoplayer. [Último acceso: Noviembre 2017].

[89] R. v. Rijn, «Java vs Javascript: Speed of Math,» [En línea]. Available: http://royvanrijn.com/blog/2012/07/java-speed-of-math/. [Último acceso: Noviembre 2017].

[90] «React Native,» [En línea]. Available: https://facebook.github.io/react-native/. [Último acceso: Noviembre 2017].

[91] «Flutter,» [En línea]. Available: https://flutter.io/. [Último acceso: Noviembre 2017].

[92] «Kivy,» [En línea]. Available: https://kivy.org. [Último acceso: Noviembre 2017].

[93] «Unity,» [En línea]. Available: https://unity3d.com/es.

[94] Qualcomm, «Trepn Power Profiler,» [En línea]. Available: https://developer.qualcomm.com/software/trepn-power-profiler. [Último acceso: Noviembre 2017].

Glosario

API

Application Programming Interface.

APK

Android Application Package.

Android

Sistema operativo para dispositivos móviles desarrollado por Google.

big.LITTLE, arquitectura

Arquitectura de cómputo heterogénea, que combina núcleos de diferentes potencias con la finalidad de ahorrar energía.

CSS

Cascading Style Sheets.

Framework

Un entorno de trabajo, que representa un conjunto de herramientas y técnicas enfocadas a resolver problemáticas particulares, evitando la reimplementación de porciones de software que resultan comunes a muchos proyectos.

GPS

Global Positioning System.

GPU

Graphics Processor Unit.

HTML

HyperText Markup Language.

HTTP

HyperText Transfer Protocol.

IDE

Integrated Development Environment.

iOS

Sistema operativo para dispositivos móviles desarrollado por Apple.

Java

Lenguaje de programación de tipado estático, compilado a un lenguaje intermedio para ser ejecutado en la Máquina Virtual de Java.

JavaScript

Lenguaje de programación interpretado, usado principalmente para aplicaciones del lado del cliente. Suele estar implementado como parte de los navegadores web.

MVC

Model View Controller.

NAND

Compuertas lógicas Not-AND, utilizadas para fabricar memorias de almacenamiento flash.

NDK

Native Development Kit.

.NET

Framework desarrollado por Microsoft, que se utiliza con el lenguaje C#.

PWA

Progressive Web Apps.

RAM

Random Access memory.

SDK

Software Development Kit.

Smartphone

Teléfono inteligente.

SoC

System on a Chip.

Swift

Lenguaje de programación compilado, de propósito general, desarrollado por Apple para iOS, MacOS y Linux.

TypeScript

Lenguaje de programación desarrollado por Microsoft, que extiende la sintaxis de JavaScript y agrega tipado estático y objetos basados en clases.

URL

Uniform Resource Locator.

Windows Phone

Sistema operativo para dispositivos móviles desarrollado por Microsoft.