UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Gestión del consumo energético en el desarrollo de aplicaciones para dispositivos móviles

Trabajo final de Master

18 Sep 2017

Master en Ingeniería y Tecnología de Sistemas de Software
Tecnología Software Multiparadigma
Orientación Profesional

Autor: Fresneda González, Sergio
Tutor: Lucas Alba, Salvador

2016/2017

Resumen

Los dispositivos móviles de uso personal (relojes y teléfonos 'inteligentes', tablets, etc.) se han convertido en pequeños ordenadores con una potencia de cómputo comparable a la de las máquinas de sobremesa de hace una década. Sin embargo, a diferencia de éstos, su funcionamiento depende de la alimentación proporcionada por una batería cuya autonomía se mide en unas pocas horas. Por este motivo, es cada vez más importante controlar con precisión el consumo eléctrico de los componentes físicos de dichos dispositivos para maximizar su rendimiento. Dicho control se extiende también al software que los dispositivos móviles ejecutan, ya que de él depende el acceso a los componentes electrónicos y electromecánicos. En este trabajo fin de máster se estudia el problema de la gestión del consumo energético en aplicaciones desarrolladas para su uso en dispositivos móviles. Partiendo de una aplicación concreta desarrollada para un fin comercial específico, se estudian los diversos factores que redundan en un mayor consumo y se proponen acciones concretas para su control a distintos niveles, desde el diseño de los algoritmos de muestreo de la señal de red que permiten la búsqueda de otros dispositivos a los que conectarse hasta el uso de unas u otras estructuras de datos en función del efecto que éstas tienen en el consumo. De esta manera, partiendo del estudio de un caso concreto se plantean acciones genéricas que pueden tenerse en cuenta en el diseño y desarrollo de software para dispositivos móviles en distintas plataformas.

Summary

Personal mobile devices like smartwatches, smartphones, tablets, etc., are just small computers with a computing performance which is similar to last-decade desktop computers. In sharp contrast, though, mobile devices depend on a battery which is often exhausted after few hours. For this reason, controlling the energy consumption of the physical components of mobile devices is more and more important to optimize its performance. Such a control also concerns the software executed in those devices, as it controls the access to the electronic and electromechanic components. In this master thesis we investigate the problem of power consumption management in software applications for mobile devices. Our starting point is a `real-world' application we use as a case study to investigate how different parts of software components contribute to power consumption. On this basis, specific control actions are proposed at different levels, including the design of sampling algorithms to get or keep other devices connected, the use of specific data structures depending on their impact in power consumption, etc. In this way, starting from a particular case-study, we enumerate a number of specific actions to be taken into account during the design and development of software for mobile devices.

Palabras clave: Dispositivos móviles, Gestión Consumo energético, Software eficiente, Arquitectura de software, Diseño software

Tabla de contenidos

1. Introducción

Un software no debe ser evaluado sólo por la cantidad de funcionalidades que tiene implementadas, sino que también se debería tener en cuenta de qué manera son optimas las funcionalidades existentes que realiza.

Es complejo definir el rendimiento desde un punto de vista técnico, ya que puede considerarse un término que quizás abarca demasiado o acota de igual manera, siempre dependiendo del contexto final en el que se encuentre. Por ejemplo, la aplicación de Facebook dispone de una interfaz y una navegación muy agradable y funcional, esto es una posible definición de buen rendimiento. Sin embargo a nivel de consumo energético esta misma aplicación es un gran ejemplo de mal rendimiento, abusando continuamente de los accesos a red y reduciendo con ello la vida de la batería.

Cuando se dice que un software tiene un gran rendimiento, se podría asumir que es debido al menor uso de memoria en sus operaciones o por ejemplo también a un correcto funcionamiento. El significado de rendimiento puede deberse a multitud de factores y con unas implicaciones todavía mayores.

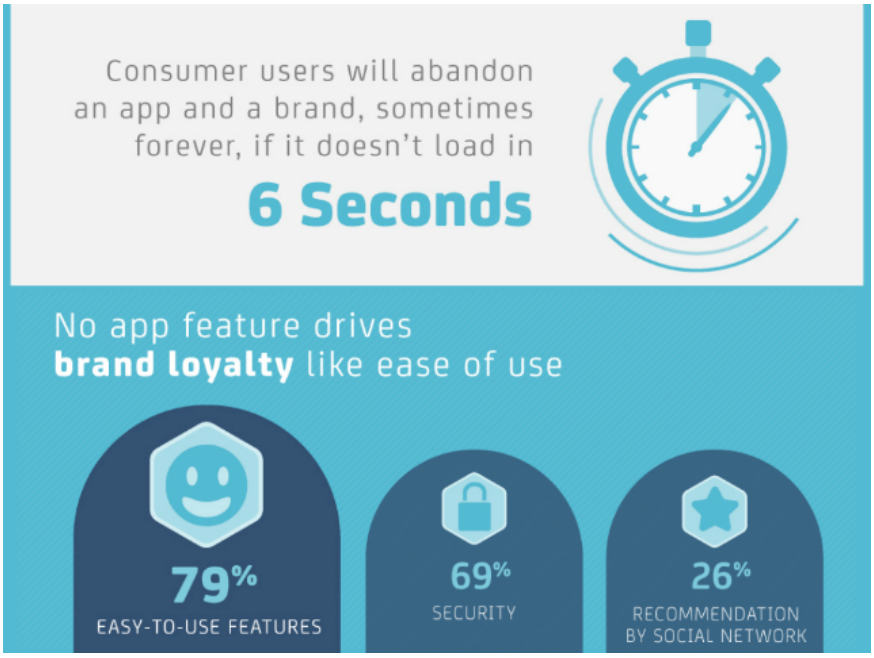

Figura 1. CA Technologies study about loyalty between consumers and businesses

En este proyecto, nos centraremos en las claves para contribuir en la mejora del rendimiento energético y cómo minimizar su consumo a nivel de código en el desarrollo de aplicaciones móviles. Trataremos los focos de consumo que se pueden encontrar durante el desarrollo de aplicaciones móviles: uso de tecnologías como el Bluetooth, el correcto uso de los estados del *hardware* o algoritmos óptimos según la necesidad. Finalmente mediante un caso de uso de un proyecto real, se explica cómo se deberían aplicar estos conocimientos de mejora de rendimiento energético, con el fin de alargar lo más posible la vida de la batería. También de qué manera se puede analizar el consumo que realiza la aplicación y de qué manera utiliza los recursos disponibles una aplicación móvil.

NOTA: A continuación aparecerán palabras en formato negrita con un asterisco al final, lo que indica que se trata de un tecnicismo y que está explicado en el glosario que se encuentra al final del proyecto. Sólo aparecerán marcadas de esta forma la primera vez que aparezcan en el documento.

1.1. Rendimiento energético

En el desarrollo de aplicaciones móviles, uno de los aspectos más importantes es el rendimiento energético, tema principal a tratar durante todo este proyecto.

El consumo energético es un factor extremadamente importante a tener en cuenta cuando se desarrolla software. El uso de estructuras de datos y algoritmos debe ser eficiente en términos de tiempo de ejecución, pero también se deben tener en cuenta otros factores.

Es muy importante mejorar el rendimiento energético, pero siempre que esto no empeore la experiencia del usuario con el software que hemos desarrollado.

Debido a la omnipresencia de los dispositivos móviles en nuestras vidas, la batería se ha convertido en uno de los factores más influyentes a la hora de elegir dispositivo. De la misma manera, el consumo energético es un factor muy importante que influye en el usuario a la hora de tener una aplicación instalada o no.
Cada componente de hardware alojado en un dispositivo móvil consume energía. El consumidor principal de energía puede ser la CPU, pero esto es dependiente del uso que se le pueda dar.

La CPU es, uno de los componentes de hardware que reduce la vida de la batería como también lo son las antenas de red, Bluetooth, GPS, cámara o la pantalla.
Una aplicación bien desarrollada debe conocer su consumo y realizarlo de manera moderada. Las aplicaciones que consumen mucha batería tienden a ser borradas por los usuarios.

1.2. Necesidad de mejora

Algunos estudios muestran que las aplicaciones cuyo tiempo de carga, oscila entre 3 y 6 segundos, son a menudo abandonadas por los usuarios, el resto que continua utilizándolas no las recomendaría a sus amigos por su mala experiencia. Vaish (2016) en su libro High Performance iOS Apps explica cómo los usuarios responden a una aplicación dependiendo de su rendimiento mediante las siguientes cifras:

- El 79% de los usuarios intenta abrir una aplicación una o dos veces más si esta falla la primera vez que se ejecuta.
- El 25% de los usuarios abandona la aplicación si esta no carga en los siguientes 3 segundos desde que se lanzó.
- El 31% de los usuarios hablará a otros usuarios sobre su mala experiencia.

Por naturaleza, todas las aplicaciones consumen energía debido al uso de distintos componentes electrónicos que permiten algún tipo de funcionalidad a la aplicación, por la generación y actualización de interfaces de usuario o por la propia ejecución de código en la CPU. Todo esto condiciona que la experiencia del usuario con una aplicación pueda ser buena o mala.

Esta experiencia de la que se habla involucra los siguientes aspectos:

Que todo fluya dentro de la aplicación: Una aplicación que no ofrece un movimiento fluido entre sus funcionalidades, indica la pobre calidad del producto. El uso intensivo de cómputo en primer o segundo plano puede ralentizar la aplicación.

Rapidez en la carga de contenidos: El usuario espera una respuesta veloz a una necesidad de información. Si esta no se produce en un corto periodo, éste buscará otras maneras de obtenerla.

El mal manejo de datos, la falta de una estructura óptima o de algoritmos eficientes para el tratamiento de la información manejada por la aplicación, podrían ser las causas.

La temperatura del dispositivo: Abusar de los recursos de manera no controlada conseguirá que el dispositivo alcance temperaturas elevadas. La falta de manejo de los ciclos de vida de cada una

Figura 2: ErgonDesk expuestas en el Espacio Fundación Telefónica (2015)

de las tecnologías utilizadas hace que éstas no pausen su consumo de recursos produciendo calor, que finalmente sufre el terminal.

El usuario quiere que su dispositivo no se quede sin batería por utilizar una aplicación y en caso de ser así, la querrá eliminar de su terminal. El comportamiento deficitario de la aplicación en alguno de los tres aspectos mencionados hará que la aplicación reduzca el nivel de vida de la batería, por lo que para cumplir este requisito, se tendrán que cumplir los tres anteriores.

1.3. Objetivos del proyecto

Este proyecto es un gran reto, debido al uso de tecnologías y componentes que sin ser desconocidas, no habían sido dominadas en su completitud hasta el momento.

Los objetivos son los siguientes:

- Como objetivo principal, mediante el caso de uso de la aplicación ErgonDesk (ver la sección 1.4), se desea conseguir una reducción de al menos un 60% del consumo energético de la aplicación en comparación con la primera versión ya publicada para iOS en la *AppStore.*

- Otro objetivo es coger la arquitectura mejorada que se ha creado en la versión para dispositivos iOS, y aplicarla en su versión para Android, con el fin de contrastar los resultados energéticos con su primera versión publicada en la *Play Store*, demostrando así una posible guía de buenas prácticas en cuanto a las tecnologías utilizadas.

1.4. Caso de estudio

El caso de estudio tratado ha sido ErgonDesk.

ErgonDesk es un escritorio de trabajo inteligente y adaptable al usuario, que permite no sólo trabajar de manera ergonómica, también mejora la capacidad de trabajo en grupo debido a su modularidad.
Dispone de una aplicación móvil que permite al usuario controlar de forma remota la ErgonDesk, recibiendo recomendaciones personalizadas para el usuario para el cambio de posturas durante la jornada laboral. Este escritorio está diseñado principalmente para el trabajo en equipo, mejorando la productividad y apoyándose en la colaboración del entorno de trabajo, además de promover una postura de trabajo mas saludable.

La elección de este proyecto como caso de estudio está motivada por mi participación en del equipo de desarrollo móvil de ErgonDesk. Esta participación ha consistido principalmente en el desarrollo de la conexión y comunicación del dispositivo móvil con la ErgonDesk, utilizando como tecnología de unión el Bluetooth.

```
    func applicationWillResignActive(_ application: UIApplication) {
        // Sent when the application is about to move from active to inactive state.
    }
    func applicationDidEnterBackground(_ application: UIApplication) {
        // Use this method to release shared resources, save user data, invalidate
timers, and store enough application state information to restore your application to
its current state in case it is terminated later.
    }
    func applicationDidBecomeActive(_ application: UIApplication) {
        // Restart any tasks that were paused (or not yet started) while the
application was inactive.
    }
```

Figura 3: Métodos de control del ciclo de vida de una aplicación en AppDelegate.Swift

Como entorno de desarrollo para la aplicación móvil se ha utilizado XCode de Apple, entorno de programación oficial para la creación de aplicaciones para iOS. En cuanto al lenguaje utilizado para el desarrollo se ha seleccionado Swift 2.2, en auge y estable durante el inicio del proyecto, posteriormente migrado a su versión 2.3 y 3.0.

La aplicación también dispone un *Back-end* propio desarrollado por el equipo web del equipo principal de desarrollo, que a su vez se comunica con otro propio e interno de la compañía PYNK SYSTEMS, que contiene la *IA* (Inteligencia Artificial) utilizada para el cálculo de posiciones que ocupa el usuario cuando utiliza la ErgonDesk.

Las principales plataformas a las que se orienta el desarrollo móvil son iOS, plataforma móvil ofrecida por Apple, y Android, sistema libre desarrollado principalmente por Google.

2. Consumo energético en dispositivos móviles

Una vez conocidas las razones que nos mueven a realizar un consumo óptimo de la batería, se deben conocer los responsables más directos que se deberían tener en cuenta.

A continuación se enumerará aquel *hardware* o tecnología, detectado como responsable del principal consumo energético en aplicaciones móviles:

2.1. Animaciones

Una regla simple a seguir: animar sólo cuando la aplicación se encuentra en primer plano. Utilizando los métodos de Swift: `applicationWillResignActive:`, `applicationDidEnterBackground:` para pausar o detener las animaciones activas y `applicationDidBecomeActive:` para reanudarlas.

Una aplicación que pasa a segundo plano dispone de un corto periodo de ejecución (si lo necesita). Cuando este tiempo se agota se crea una copia en memoria del estado actual de la aplicación y se suspende completamente. En estado de suspensión la aplicación no puede realizar ninguna acción a no ser que disponga de privilegios como el uso de voz sobre ip (VOIP) o streaming de audio.

2.1.1. Buenas prácticas

Las siguientes buenas prácticas se han extraído de la web de desarrolladores de Apple[1], que se puede encontrar en la sección de referencias del proyecto.

EVITAR GRÁFICOS Y ANIMACIONES PERSONALIZADOS

Si la aplicación sólo utiliza ventanas y controles estándar, no debería dar problemas energéticos, ya que las **API*** del sistema están diseñadas para maximizar la eficiencia energética. Sin embargo, si la aplicación dispone de ventanas y controles personalizados, se debería de comprobar que se ejecutan de manera eficiente. La aplicación no debe actualizar el contenido innecesariamente, como en áreas oscurecidas en la pantalla, o mediante el uso excesivo de animaciones.

El dibujo excesivo o ineficiente puede sacar a los recursos del sistema del estado de baja potencia o evitar que se apaguen, lo que acaba siendo un uso significativo de la energía.

Directrices para optimizar las actualizaciones de contenido:

- Reducir el número de vistas que utiliza la aplicación.

- Reducir el uso de la opacidad, como las vistas que muestran un desenfoque translúcido. Si es necesario utilizar opacidad, se debe evitar utilizarla en el contenido que va cambia con frecuencia. El coste de energía en el dibujado de estas vistas es grande, ya que tanto la vista de fondo como la vista translúcida deben actualizarse siempre que se cambie el contenido.

- Eliminar el dibujado cuando la aplicación o su contenido no sea visible, como cuando el contenido de la aplicación se encuentra debajo de otras vistas, recortadas o fuera de pantalla.

- Utilizar velocidades de fotogramas inferiores para animaciones siempre que sea posible. Una velocidad de fotogramas alta puede tener sentido en un juego, pero una velocidad de fotogramas más baja puede ser suficiente para una pantalla de menú. Es importante utilizar una velocidad de fotogramas alta sólo cuando la experiencia del usuario lo solicite.

```
//Views
let spriteKitGame : SKView?
let sceneKitGame : SCNView?
let metalKitGame : MTKView?
let viewController : UIView?

//Enumerable kind of app
enum kindOfApp{
    case twoDimensionsGame
    case threDimensionsGame
    case higlyInmersiveGame

    case genericApp
}

//Rich initialization with comprensive String
extension kindOfApp : RawRepresentable{
    typealias RawValue = String

    init?(rawValue: RawValue) {
        switch rawValue {
        case "2d"  : self = .twoDimensionsGame
        case "3d"  : self = .threDimensionsGame
        case "inmersive" : self = .higlyInmersiveGame
        case "app" : self = .genericApp
        default : return nil
        }
    }

    var rawValue : RawValue{
        switch self{
        case .twoDimensionsGame: return "2d"
        case .threDimensionsGame : return "3d"
        case .higlyInmersiveGame : return "inmersive"
        case .genericApp : return "app"
        }
    }
}

//Example variable with the kind of app we gonna develop
let myApp = kindOfApp.init(rawValue: "3d")

//Depend kind of application will be applied initialization of view and his framerate
switch (myApp?.rawValue)! as String{

case "2d": //2D game with SpriteKit framework and 60fps becouse is a game and we need the best
user experience
    spriteKitGame = SKView()
    spriteKitGame?.preferredFramesPerSecond = 60

case "3d": //3D game with SceneKit framework and 60fps becouse is a game and we need the best user
experience
    sceneKitGame = SCNView()
    sceneKitGame?.preferredFramesPerSecond = 60

case "inmersive": //Metal game with MetalKit framework and 60fps becouse is a game and we need the
best user experience
    metalKitGame = MTKView()
    metalKitGame?.preferredFramesPerSecond = 60

case "app": //For a generic app is not needed use a framework for game develop.
    viewController = UIView()

default: //In other case this app is Unknown
    break
}
```

Figura 4: Dependiendo de la aplicación a desarrollar se puede utilizar un framework de juegos u otro

- Utilizar una frecuencia de fotogramas constante al realizar una animación. Por ejemplo, si la aplicación muestra 60 fotogramas por segundo, mantener esa velocidad de fotogramas a lo largo de la animación.

- Evitar utilizar múltiples velocidades de fotogramas a la vez en la pantalla. Por ejemplo, un personaje de un juego moviéndose a 60 fotogramas por segundo, mientras que las nubes en el

cielo se mueven a 30 fotogramas por segundo. Se debería utilizar la misma velocidad de fotogramas para ambos, incluso si significa aumentar una de las velocidades de fotogramas.

- Utilizar los **frameworks*** recomendados al desarrollar juegos. Estos frameworks están optimizados para proporcionar un gran rendimiento y una eficiencia energética óptima:
  - Utilizar SpriteKit para juegos 2D.
  - Utilizar SceneKit para juegos 3D casuales.
  - Utilizar Metal para juegos de gran inmersión.

En la *figura 4* se han aplicado estos conceptos con lenguaje Swift.

DETENER EL DIBUJADO DE LA INTERFAZ AL REPRODUCIR VÍDEO EN PANTALLA COMPLETA
Si las capas de interfaz de la aplicación que no se encuentran visibles mientras un vídeo que reproduce a pantalla completa continúan realizando dibujados mientras no son visibles, pueden degradar la optimización de la aplicación aumentando el uso de recursos como la **GPU***.

2.2. Bluetooth

Bluetooth puede ser también uno de los grandes problemas de consumo energético si no se utiliza con cautela. Si el alcance de la aplicación está bien definido y conocemos el tipo de dispositivos con el cual vamos a interactuar, se puede evitar un consumo innecesario.

Inicialmente, cuando se realiza una búsqueda de dispositivos, podemos utilizar de la mano del

```swift
// Stop scan queue
private let stopScanOperationQueue : OperationQueue = {
    let queueOp = OperationQueue()
    queueOp.maxConcurrentOperationCount = 1
    return queueOp
}()

func findDevices(with services: [CBUUID]? = nil, options:  [String:Any]? = nil){
    self.centralManager.stopScan()
    if !self.centralManager.isScanning(){
        self.centralManager.scanForPeripherals(withServices: services, options: options)
        self.addCancelScanOperationForInactive()
   }
}

func addCancelScanOperationForInactive(){
    // Stop all posible operations in queue
    self.stopScanOperationQueue.cancelAllOperations()

    // Create a new block of operations
    let operation = BlockOperation {}
    operation.addExecutionBlock {
        // Sleep the operation for 30 seconds
        // if the operation isn't cancelled in 30 seconds, it call to stopScan
        Thread.sleep(forTimeInterval: 30)
        guard !operation.isCancelled else { return }
        self.centralManager.stopScan()
    }
    self.stopScanOperationQueue.addOperation(operation)
}
```

Figura 5: Detención de búsqueda por inactividad en Swift

`CBCentralManager` un método llamado `scanForPeripherals`. Este método nos permite buscar dispositivos cercanos de diferentes formas.

En iOS el framework* `CoreBluetooth` permite a la aplicación tener el control y ser responsable de implementar la mayoría de los aspectos del rol central, como el descubrimiento y la conectividad de dispositivos, y explorar e interactuar con los datos de un periférico remoto.

```
    func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral,
advertisementData: [String : Any], rssi RSSI: NSNumber) {
        guard !self.peripheralList.contains(peripheral) else {
            return
        }

        let devicesArray = self.peripheralList.filter({$0.identifier == peripheral.identifier}
        if devicesArray.isEmpty {
            self.addCancelScanOperationForInnactive()
            self.peripheralList.append(peripheral)
        }
    }
```

Figura 6: Añadir una nueva operación al detectar un periférico en Swift

Minimizar la transmisión de datos es especialmente importante cuando se desarrolla una aplicación para un dispositivo iOS, porque tiene un efecto adverso en la duración de la batería de un dispositivo.

2.2.1. Buenas prácticas

BUSCAR DISPOSITIVOS SOLO CUANDO SEA NECESARIO

Cuando llama al método `scanForPeripheralsWithServices:options:` de la clase `CBCentralManager` para descubrir los periféricos remotos, el dispositivo utiliza la antena Bluetooth para escuchar dispositivos hasta que le indique explícitamente que deje de hacerlo.

A menos que necesite descubrir más dispositivos, se debe dejar de buscar otros dispositivos después de haber encontrado uno con el que desea conectarse. Se utiliza el método `stopScan` de la clase `CBCentralManager` para detener el escaneado de otros dispositivos.

La búsqueda se puede realizar sin parámetros, es decir sin filtrar qué dispositivos vamos a encontrar, esto lleva un tiempo de escaneo del entorno más prolongado lo que conlleva un mayor consumo energético. No obstante, si conocemos que tipo de dispositivos vamos a buscar (mediante sus servicios), podemos asignarlo a la búsqueda y reducir el coste en gran medida.

Es muy necesario controlar el tiempo que estamos utilizando en buscar dispositivos. Una buena práctica sería comprobar si durante un periodo de tiempo no se ha detectado ningún dispositivo nuevo, si es así, se puede detener la búsqueda.

```
self.centralManager.scanForPeripherals(withServices: nil, options:
[CBConnectPeripheralOptionNotifyOnDisconnectionKey:true])
```

Figura 7: Permitir claves duplicadas en Swift

En el ejemplo de la *Figura 5* para controlar el tiempo de búsqueda, primeramente se ha creado una cola de operaciones `stopScanOperationQueue`, la cual permite encolar operaciones que detengan la búsqueda de dispositivos.
Cada vez que se lanza una búsqueda con `findDevices:services:options:`, se lanza una operación de cancelación. El método `addCancelScanOperationForInactive` primero

cancela cualquier operación encolada, a continuación se crea un nuevo bloque de operación el cual esperará durante 30 segundos a ser cancelada; en caso de no ser cancelada lanza una detención de la búsqueda de dispositivos.

Hasta ahora la búsqueda se limita a 30 segundos, si no se detectan dispositivos nuevos en los siguientes 30 segundos a la creación del bloque se cancelará la búsqueda. Para controlar la

```
    let k_SendDataServiceUUID   : CBUUID = CBUUID(string: "AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA")
    let k_ReadDataServiceUUID   : CBUUID = CBUUID(string: "BBBBBBBB-BBBB-BBBB-BBBB-BBBBBBBBBBBB")

    let k_ServicesCollection    : [CBUUID] = [k_SendDataServiceUUID,
                                              k_ReadDataServiceUUID]

    /// Invoked when a connection is successfully created with a peripheral.
    func centralManager(_ central: CBCentralManager, didConnect peripheral: CBPeripheral) {
        peripheral.delegate = self

        //Call to discoverServices with a collection of services
        peripheral.discoverServices(k_ServicesCollection)

        self.centralManager.stopScan()
    }

    func peripheral(_ peripheral: CBPeripheral, didDiscoverServices error: Error?) {

        //Get comparable UUIDs from a constant with all needed services
        let templateServices = k_ServicesCollection.flatMap{$0.uuidString}

        //Get comparable UUIDs from the current peripheral
        let servicesUUIDs = peripheral.services.flatMap{$0.uuid.uuidString}

        //Compare both lists of identifiers
        let filteredServices = servicesUUIDs.filter{templateServices.contains($0)}

        //If this finds the services, it will obtain the characteristics
        if  filteredServices.count == templateServices.count{
            let _ = services.map{ peripheral.discoverCharacteristics(k_CharacteristicsCollection,
for: $0)}
        }
    }
```

Figura 8: Buscar servicios y características necesarios en Swift

detección de nuevos periféricos descubiertos también se llamará a este método desde el **delegado*** del `centralManager:didDiscoverPeripheral:advertisingData:RSSI:`, encargado de mostrar los dispositivos encontrados al resto de la lógica de la aplicación.

Cuando se detecta un periférico, se comprueba si existe en la lista de periféricos encontrados. En caso de ser nuevo, crea una nueva operación de cancelación de la búsqueda y añade el dispositivo a la lista de los periféricos conocidos, en caso contrario se continua con la búsqueda el tiempo restante que le quede a la operación.

```
    func peripheral(_ peripheral: CBPeripheral, didDiscoverCharacteristicsFor service: CBService,
error: Error?) {
        let characteristics = service.characteristics

        //Subscribe for all founded characteristics
        for characteristic in characteristics {
            peripheral.setNotifyValue(true, for: characteristic)

        }
    }
```

Figura 9: Subscripción a las características en Swift

Los dispositivos Bluetooth remotos pueden enviar múltiples paquetes por segundo para anunciar su presencia a los dispositivos de escucha. Cuando se escanea dispositivos con el método `scanForPeripheralsWithServices:options:` (ejemplo 2.2.1-1), el comportamiento predeterminado del método es agrupar paquetes de descubrimiento de un periférico en un único evento de descubrimiento, es decir, el dispositivo de escucha llama al método `centralManager:didDiscoverPeripheral:advertisingData:RSSI:` del delegado sólo una vez por cada nuevo periférico que descubre, independientemente de cuántos paquetes de

```
    func peripheral(_ peripheral: CBPeripheral, didUpdateValueFor characteristic:
CBCharacteristic, error: Error?) {

        print("Characteristic updated: \(characteristic).\n Error?: \
(error!.localizedDescription)")

    }
```

Figura 10: Obtención de la notificación de una característica en Swift

descubrimiento reciban de cada uno.

Se puede especificar la constante `CBCentralManagerScanOptionAllowDuplicatesKey` como una opción de exploración al llamar al método `scanForPeripheralsWithServices:options:`. Al hacerlo, un evento de descubrimiento se genera cada vez que el dispositivo recibe un paquete de descubrimiento. Desactivar el

```
    /// Subscribe or Unsubscribe from all services available on the peripheral received in params
    private func ed_subscribeToServices(from peripheral: CBPeripheral, subscribe: Bool){
        guard let services = peripheral.services else{
            print("Peripheral \(peripheral.identifier.uuidString) dont have services available")
            return
        }

        // Subscribe in all characteristic of BLE device
        // Service  -> Characteristic 1
        //          -> Characteristic 2
        //         ...
        //          -> Characteristic n

        for service in services{
            print("Subscribing to characteristics from \(service.uuid.uuidString)")
            if let characteristics = service.characteristics{
                for characteristic in characteristics{
                    if characteristic.isNotifying{
                        peripheral.setNotifyValue(subscribe, for: characteristic)
                    }
                }
            }
        }
    }

    /// Allow disconnect from actual BLE device
    /// When user want to disconnect from a device, this unsubscribe from his services
    func ed_disconnect(from device: CBPeripheral? = nil){
        guard self.ed_deskStatus() != .disconnected && self.ed_deskStatus() !=  .disconnecting
else{
            print("Refusing to disconnect \(self.ed_deskStatus().rawValue)")
            return
        }
        if let peripheral = self.actualPeripheral{
            print("\(NSDate()): Disconnecting from peripheral \(peripheral) status: \
(self.ed_deskStatus().rawValue)")

            self.ed_subscribeToServices(from: peripheral, subscribe: false)
            self.ed_disconnect(from: device)
        }
    }
```

Figura 11: Desconexión del dispositivo y cancelación de la subscripción a las características en Swift

comportamiento predeterminado puede resultar útil en algunos casos, como iniciar una conexión a un periférico en función de la proximidad de éste. La especificación de esta opción de análisis puede tener un efecto adverso en la duración de la batería y el rendimiento de la aplicación. Por lo tanto, esta opción de exploración se debe utilizar sólo cuando sea necesario para cumplir con un caso de uso particular (*Figura 7)*.

OBTENER SÓLO LOS SERVICIOS Y CARACTERÍSTICAS NECESARIOS
Los dispositivos periféricos proporcionan servicios relacionados con la realización de funciones específicas, como la ErgonDesk que ofrece un servicio para recuperar información de la altura e inclinación de la mesa. Estos servicios incluyen características o atributos.

Un periférico puede tener uno o más servicios y características de las necesarias para cumplir con un caso de uso específico con la aplicación. Por eso, se debe buscar y descubrir los servicios y características específicos que necesita la aplicación. En iOS se puede hacer proporcionando los identificadores únicos (UUID, del inglés Universally Unique Identifier) específicos a los métodos `discoverServices:` y `discoverCharacteristics:forService:` de la clase `CBPeripheral` (*Figura 8)*.

SUBSCRIBIRSE A LAS NOTIFICACIONES EN LUGAR DE REALIZAR LECTURAS CONTINUADAS
La aplicación no tiene porqué saber cuándo cambiará el valor de la característica de un servicio del dispositivo al que está conectado. Una forma de conocer este valor podría ser consultar repetidamente el dispositivo (por sondeo) para detectar cambios. Esto multiplica al menos por dos el número de señales que se le mandan al dispositivo, ya que por cada escritura se debería realizar una lectura para confirmar y actualizar el nuevo valor en la aplicación.
Existe una manera más eficiente, registrarse para recibir notificaciones cuando se producen cambios en las características y servicios del periférico (*Figura 9)*.

En la *Figura 10* se observa que para subscribirse al valor de una característica se le pasarán a `setNotifyValue:forCharacteristic:` dos parámetros: primero el valor **Booleano*** (true o false) que indica si se quiere subscribir a las notificaciones del periférico, el segundo parámetro indica sobre qué característica del periférico se pretende realizar la subscripción. Cuando cambia el valor de la característica, el periférico llama al método `peripheral:didUpdateValueForCharacteristic:error:` de su objeto delegado.

DESCONECTAR EL DISPOSITIVO CUANDO YA NO SEA NECESARIO
Para evitar que la aplicación utilice el Bluetooth sin necesidad, se debe realizar una desconexión del periférico como se aprecia en la *Figura 11*. Cancelar cualquier suscripción de notificación pasando un valor de Booleano *false* al método `setNotifyValue:forCharacteristic:` de la clase `CBPeripheral`. A continuación, proceder a la desconexión del dispositivo llamando al método `cancelPeripheralConnection:`.

2.3. CPU

La Unidad Central de Proceso (CPU, del inglés Control Processing Unit) es el hardware principal utilizado por una aplicación. El procesador que se puede encontrar en los actuales dispositivos pueden ser dual-core (dos núcleos de proceso), tri-core (tres núcleos), o superior.

En la siguiente tabla extraída de GeekBench[2] se muestran algunos ejemplos de dispositivos trabajando con uno o varios núcleos y como esto se ve reflejado en su potencia.

Dispositivo	Procesador	Núcleos	Bits	Frecuencia Reloj	Benchmark Single-Core	Benchmark Multi-Core
iPad Pro (10.5”)	A10X Fusion	6	64Bits	2.3GHz	3878	9179
iPad Pro (12.9” 2ª Gen)	A10X Fusion	6	64Bits	2.3GHz	3873	9146
iPhone 7 Plus	A10 Fusion	4	64Bits	2.3GHz	3331	5538
iPhone 7	A10 Fusion	4	64Bits	2.3GHz	3325	5523
iPad Pro (12.9” 1º Gen)	A9X	2	64Bits	2.3GHz	3030	4973
iPhone SE	A9	2	64Bits	1.8GHz	2399	4075
iPhone 6s Plus	A9	2	64Bits	1.8GHz	2396	4058
iPhone 6s	A9	2	64Bits	1.8GHz	2311	3920

Tabla 1: Características y resultados de varios dispositivos móviles

Cuanto más cómputo realiza una aplicación, mayor es su consumo. Cada generación de dispositivos consume mucha menos energía por el uso de la mismas operaciones.

La cantidad de cómputo utilizada depende de varios factores:

```swift
func quicksort<T: Comparable>(_ a: [T]) -> [T] {
    guard a.count > 1 else { return a }

    let pivot = a[a.count/2]
    let less = a.filter { $0 < pivot }
    let equal = a.filter { $0 == pivot }
    let greater = a.filter { $0 > pivot }

    return quicksort(less) + equal + quicksort(greater)
}
```

Figura 12: Algoritmo Quicksort en Swift

- Número de veces que se realiza una actualización. El resultado de una operación que genera valores de manera continua o de una vista de la Interfaz de Usuario (UI, del inglés User Interface) que muestra información actualizada de un dato cambiante. Por ejemplo una **notificación *push**** recibida por una aplicación de chat actualiza una serie de valores con cada mensaje recibido en la aplicación. Estos valores cambiantes resultado de la notificación pueden hacer que la UI necesite actualizarse de manera continuada.

- Procesado de datos (ej: tratamiento de un fichero de datos). Habitualmente la comunicación entre aplicaciones y el servidor se realiza mediante ficheros ordenados. Éste orden de los datos sigue un estándar, como puede ser JSON (JavaScript Object Notation) o XML (Extensible Markup Language).

- Algoritmos y estructuras usadas para el proceso de datos. El proceso de realizar una operación puede variar su coste de CPU dependiendo de cómo están organizados los datos y como se abordan. El variado grupo de algoritmos de ordenación (QuickSort, MergeSort,

BubbleSort…)[3] es un ejemplo de como realizar una tarea de distintas formas pero con el mismo resultado puede tener un distinto coste de CPU.

- Tamaño de los datos a procesar. Dependiendo del tamaño de la pantalla del dispositivo el software tiene que mostrar más o menos información en una sola vista, lo que significa más datos que se tienen que manejar y que pueden ser de distinto tamaño.

No hay una única regla a seguir para reducir el tiempo de procesamiento de la aplicación.

2.3.1. Buenas prácticas

A continuación se detallarán una ser de buenas prácticas (algunas obtenidas del libro de Vaish High Performance iOS Apps[4] y transcritas en Swift para este proyecto) que pueden mejorar el consumo energético de la CPU.

2.3.1.1. Algoritmos

Utilizar el algoritmo más optimo para cada tipo de escenario.

Muchas aplicaciones puede realizar tareas de ordenamiento. Según el autor Vaish, es preferible realizar una inserción ordenada que una mezcla de valores ordenados cuando la lista de elementos es inferior o igual a 43. Cuando el número de entradas es mayor a 286 es preferible utilizar *Quicksort*. Es preferible utilizar el algoritmo de *Quicksort* de doble pivote sobre el algoritmo tradicional de un solo pivote. Obtenido de RayWenderlich[5].

2.3.1.2. Minimizar el procesado por parte del cliente

Si una aplicación recibe datos de un servidor, se deberá minimizar la necesidad de procesado de datos por parte del cliente.

Para el funcionamiento de una aplicación cliente, a menudo es necesario que obtenga información de un servidor de internet utilizando una API. Para evitar un complejo procesado por parte del cliente, la API sirve los datos que utilizará la aplicación ya tratados y preparados para ser mostrados al usuario final por pantalla. El fin de esta práctica es ahorrar recursos en la preparación de los datos que se van a mostrar al usuario, por parte del dispositivo.

Un ejemplo NO óptimo podría ser una aplicación del tiempo que muestra la información meteorológica actual en la ciudad de Valencia, España.
Esta aplicación obtiene la información a través de una API. Cuando la aplicación comienza su ejecución le pide a la API la información del tiempo, esta API no trata los datos por lo que sirve a la aplicación la información meteorológica de todas las ciudades de Europa, la información meteorológica está formada por un boletín agrometeorológico, un balance hídrico y el nivel de radiación ultravioleta. Con toda esta información la aplicación debe realizar un filtrado entre todas las ciudades de Europa y realizar una predicción del tiempo que va a hacer en la ciudad.

La versión óptima del ejemplo anterior sería la siguiente. Cuando la aplicación comienza su ejecución le pide a la API la información del tiempo de la ciudad deseada, la API responde con la información meteorológica de la ciudad que se ha solicitado, esta información está compuesta por la temperatura actual/máxima/mínima, el tiempo actual (lluvia, sol, nubes…), humedad, dirección del viento y velocidad en Km/h.

2.3.1.3. Optimización de la Compilación Anticipada (AOT)

La Compilación Anticipada (AOT, del inglés Ahead Of Time), es la generación previa del contenido que se va a utilizar en la aplicación. Esta creación de contenido se realiza durante el tiempo de compilación, reduciendo el tiempo de espera para el usuario cuando utiliza la aplicación. El problema que genera el abuso de este tipo de compilación es la carga que se genera en el procesador y la memoria, teniendo que almacenar todo el contenido pre-cargado se utilice o no se utilice.

El procesamiento Justo A Tiempo (JIT del inglés Just In Time) fuerza al usuario a esperar para realizar una operación.

La cantidad exacta de AOT requerido es dependiente de cada aplicación y dispositivo.

Si se realizan tareas de dibujado de tablas de contenidos, no es recomendable procesar todas las entradas de la tabla. Dependiendo del tamaño de cada entrada, el dispositivo puede dibujar N registros. Cuando el usuario intenta acceder a un registro de la tabla inferior de manera rápida, no es necesario cargar todos los registros de manera inmediata, si no que se puede realizar un tiempo de espera hasta que el usuario haya reducido su velocidad de navegación por la tabla. El tiempo de espera dependerá del tipo de contenido de las entradas de la tabla (texto, imágenes,

```
    let manager : CLLocationManager = CLLocationManager()

    override func viewDidLoad() {
        super.viewDidLoad()
        self.manager.delegate = self
    }

    //Action for enable location services
    @IBAction func enableLocationServices(_ sender: Any){
        self.manager.distanceFilter = kCLDistanceFilterNone
        self.manager.desiredAccuracy = kCLLocationAccuracyBest

        self.manager.startUpdatingLocation()
    }

    //CLLocationManager delegate
    func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation])
{
        print(locations.last!)
    }
```

Figura 13: Inicialización y obtención de la posición del usuario en Swift

vídeos…)

2.4. GPS y localización

Es importante saber que los servicios de localización del GPS (del inglés Global Positioning System) y WiFi requieren de un consumo intenso de la batería.

Según Wikipedia, GPS es un sistema para determinar la posición de un objeto sobre la tierra. Una variante de este sistema es el GPS diferencial que permite localizar un objeto con una precisión de centímetros, cuando el sistema habitual utilizado por los dispositivos con esta tecnología tiene una precisión de unos pocos metros. Este sistema de posicionamiento tiene un total de 24 satélites alrededor de la tierra, posicionados a 20 200km de altura sobre el nivel del mar.

El cálculo de la localización del usuario mediante GPS depende principalmente de dos factores:

- Tiempo. Cada transmisión de GPS es única y se genera con un número pseudo-aleatorio de 1023-bit cada milisegundo, con un ratio de datos de 1024 Mbps. El chip receptor de GPS debe alinearse correctamente con el tiempo actual del satélite.

- Frecuencia. El receptor GPS debe calcular cualquier señal relativa al movimiento entre el satélite y el receptor.

Enlazar de manera completamente correcta con el satélite tarda aproximadamente 30 segundos, y la localización se obtiene por cualquier satélite que se encuentre en el rango del receptor. Cuantos más satélites se encuentren en este rango, más exacta será la localización.

Calcular la posición de esta manera requiere de un uso intenso de la CPU y del GPS, cuando ambos trabajan juntos pueden consumir la batería muy rápidamente.

NOTA: Las siguientes buenas prácticas han sido inspiradas por las utilizadas en el libro de Vaish.

2.4.1. Buenas prácticas

Para entender la complejidad de la localización del GPS, veremos cómo hay que utilizarlo pero con cuidado, particularmente si se desea desarrollar una aplicación plenamente basada en mapas.

A continuación se mostrarán qué buenas prácticas hay que seguir para minimizar el consumo energético en lenguaje Swift.

Como se puede observar en la *figura 13*, hay dos parámetros que tienen un papel muy importante cuando se llama a `startUpDatingLocation`:

- `distanceFilter`

```
    //Init Location manager object
    self.locationManager = CLLocationManager()
    self.locationManager.delegate = self

    //Request location authorization when the user is on the app
    self.locationManager.requestWhenInUseAuthorization()

    self.getLocation()

    func getLocation(){
        //Location services enabled for this app?
        if CLLocationManager.locationServicesEnabled(){
            //Get location
            self.locationManager.requestLocation()
        }
    }

    func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation])
{
        print("New Latitude: \((locations.last?.coordinate.latitude)!)")
        print("New Longitude: \((locations.last?.coordinate.longitude)!)")
    }

    func locationManager(_ manager: CLLocationManager, didFailWithError error: Error) {
        print(error.localizedDescription)
    }
```

Figura 14: Obtener ubicación de manera poco precisa en Swift

El filtro de distancia hará que el manejador notifique al delegado acerca de los eventos `locationManager:didUpdateLocations` sólo si el dispositivo se ha movido una distancia mínima. La distancia es representada en unidades del SI (metros).

Esto no ayudará a reducir el consumo del GPS, pero reducirá indirectamente el consumo de la CPU ya que salvo que la distancia entre la primera toma de posición y la última sean superiores al filtro indicado, esta última posición no será procesada completamente.

- `desiredAccuracy`

El parámetro de la precisión tiene un impacto directo en el número de antenas que se utilizan, y con ello en el consumo de la batería. Se debe elegir un nivel de precisión basado en las necesidades de la aplicación. La precisión de la señal, se define con las siguientes constantes:

- `kCLLocationAccuracyBestForNavigation`
  Se trata del mejor nivel de precisión para usos de navegación.

- `kCLLocationAccuracyBest`
  Es el mejor nivel de precisión para un dispositivo.

- `kCLLocationAccuracyNearestTenMeters`
  Precisión de hasta 10 metros. Se utiliza cuando no es necesario conocer cada metro recorrido por el usuario.

- `kCLLocationAccuracyHundredMeters`

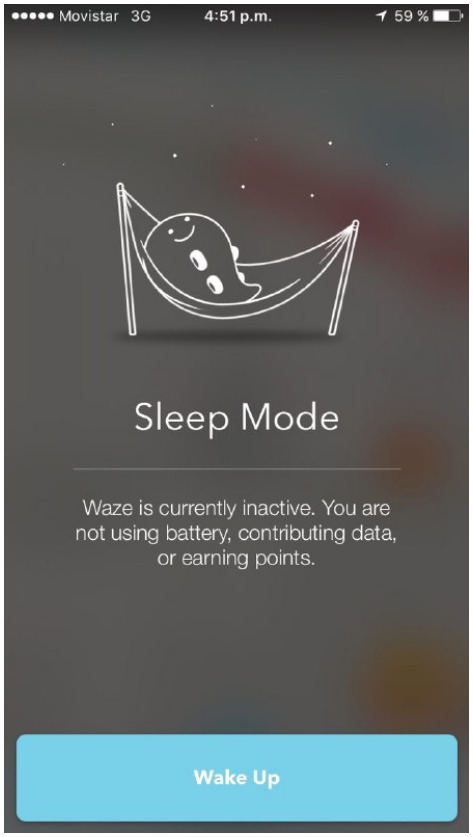

Figura 15: Aplicación Waze en modo suspensión

  Precisión de hasta 100 metros.

- `kCLLocationAccuracyKilometer`
  Precisión de hasta 1000 metros. Útil cuando se desea calcular la distancia aproximada entre dos puntos que pueden estar a cientos de kilómetros de distancia.

- `kCLLocationAccuracyThreeKilometers`
  Precisión de hasta 3000 metros.

SOLICITAR ACTUALIZACIONES DE UBICACIÓN RÁPIDA

Si la aplicación no requiere de una ubicación demasiado exacta, es mejor llamar al método `requestLocation` del objeto del `locationManager`. De este modo, se detendrán automáticamente los servicios de ubicación una vez que se haya cumplido la solicitud, dejando el hardware de posicionamiento GPS apagado si no se utiliza en otro

```
    var locationManager : CLLocationManager?

    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        self.locationManager = CLLocationManager()

        return true
    }

    func applicationDidEnterBackground(_ application: UIApplication) {
        self.locationManager?.stopUpdatingLocation()
        self.locationManager?.startMonitoringSignificantLocationChanges()
    }

    func applicationWillEnterForeground(_ application: UIApplication) {
        self.locationManager?.stopMonitoringSignificantLocationChanges()
        self.locationManager?.startUpdatingLocation()
    }
```

Figura 16: Control de la monitorización según el estado de la aplicación en Swift

lugar. Las actualizaciones de ubicación solicitadas de esta manera se entregan mediante una devolución de llamada al método `locationManager:didUpdateLocations:`, que debe implementar la aplicación (*Figura 14)*.

ES IMPORTANTE MANTENER APAGADAS AQUELLAS FUNCIONALIDADES QUE NO SE UTILIZAN

Es importante decidir cuándo se necesita realizar el seguimiento de los cambios de ubicación. Invocar a `startUpdatingLocation` cuando sea necesario realizar un seguimiento y `stopUpdatingLocation` para detenerlo cuando no sea necesario. Ambos métodos provenientes del `CLLocationManager` ya vienen implementados en la librería `MapKit` de Swift.

Un ejemplo podría ser una aplicación como iMessages (chat nativo en iOS), que permite compartir la ubicación con otros usuarios. Para compartir la ubicación podrían existir dos opciones:

- Enviar la situación actual del usuario, obtener la posición actual e inmediatamente desactivar los servicios de posicionamiento.
- Compartir la posición durante un periodo de tiempo mediante una corta monitorización del usuario (velocidad, dirección…). Se podría prever cada qué periodo de tiempo sería necesario obtener la posición del usuario; así evita el uso constante del GPS.

El seguimiento de la ubicación debería desactivarse si la aplicación no está en primer plano o si el usuario no está utilizándola.
Una solución aún mejor es dar al usuario final una opción para desactivar características no esenciales. Por ejemplo, la conocida aplicación de mapas colaborativos Waze que ofrece la opción de desactivar todas las actividades de la aplicación si el usuario no está utilizando la aplicación.

UTILIZAR LA RED SÓLO SI ES ESENCIAL.

Por eficiencia energética, iOS desactiva el hardware de red tanto como sea posible. Cuando una aplicación necesita establecer una conexión de red, iOS aprovechará esta oportunidad para

```swift
    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        self.locationManager = CLLocationManager()
        //If app was restarted due to location changes after it was killed for lack of resources
        if let options = launchOptions, options[UIApplicationLaunchOptionsKey.location] != nil{
            //Start monitoring for location changes. Otherwise, start monitoring at a later
appropriate time.
            self.locationManager?.startMonitoringSignificantLocationChanges()
        }

        return true
    }
```

Figura 17: Reinicio de monitorización si la aplicación es finalizada en Swift

permitir que las aplicaciones de fondo compartan esta sesión de red, de modo que puedan procesarse los eventos de baja prioridad, como notificaciones push, recuperación de correo electrónico, etc.

El resultado es que siempre que la aplicación se conecta a la red, el hardware de red permanecerá activo durante varios segundos después de que la aplicación termine con la conexión. Cada una de estas ráfagas de tráfico de red puede consumir un punto porcentual completo de duración de la batería.
Para minimizar este problema, el software necesita hacer un uso más económico de la red. Debería tratar de procesar el acceso a la red en ráfagas periódicas en lugar de un flujo de datos continuamente activo para que el hardware de red se pueda desactivar.

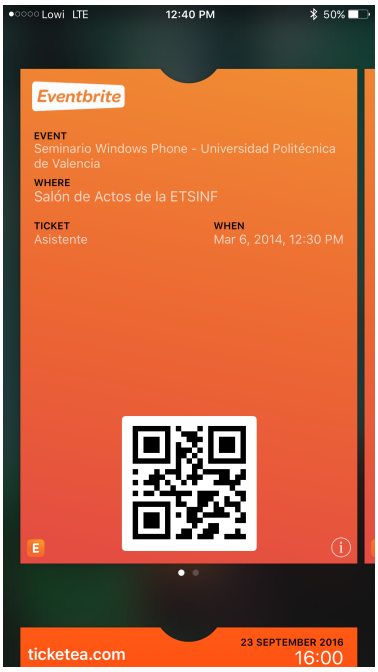

Figura 17: PassBook, la aplicación nativa de iOS para la lectura de distintos tipos de código

SERVICIOS DE UBICACIÓN EN SEGUNDO PLANO
En Swift, `CLLocationManager` proporciona un método alternativo para escuchar las actualizaciones de ubicación. `StartMonitoringSignificantLocationChanges` ayuda a seguir el movimiento para distancias más largas. El valor exacto se determina internamente y es independiente del filtro de distancia.

En la *figura 16* se realiza un seguimiento del movimiento cuando la aplicación entre en segundo plano. Un comportamiento típico sería utilizar `startMonitoringSignificantLocationChanges` cuando la aplicación está en segundo plano (`applicationDidEnterBackground`) y `startUpdatingLocation` cuando va a estar en primer plano (`applicationWillEnterForeground`).

USO DE TIMERS, THREADS Y SERVICIOS DE UBICACIÓN

```
NotificationCenter.default.addObserver(
      CFRunLoopObserver.self,
      selector: Selector((“powerMode:")),
      name: NSNotification.Name.NSProcessInfoPowerStateDidChange,
      object: nil)
```

Figura 18: Registrar cambios en el estado de batería en Swift

Los *timers* o *threads* se suspenden cuando se pone la aplicación en suspensión. Pero si se ha solicitado que la ubicación se actualice cuando la aplicación está en segundo plano, esta se despertará durante un tiempo infinitesimal cada vez que se recibe datos nuevos. Durante este

```
func powerMode(){
    if ProcessInfo.processInfo.isLowPowerModeEnabled {
        // Low Power Mode is enabled. Start reducing activity to conserve energy.
    } else {
        // Low Power Mode is not enabled.
    }
}
```

Figura 19: Comprobación del estado de activación del modo ahorro de batería en Swift

corto periodo, los *threads* y *timers* también cobrarán vida nuevamente.

El peor de los casos sería si se realiza cualquier operación de red durante ese periodo, ya que esto además activará todas las antenas relacionadas con la carga de datos (es decir, WiFi, LTE / 4G / 3G).

REINICIAR DESPUÉS DE QUE LA APLICACIÓN HAYA SIDO FINALIZADA

Por último, pero no por ello menos importante, es posible que la aplicación se destruya si se realiza en segundo plano y otra aplicación necesita recursos. Si ese es el caso, cada vez que ocurre un cambio de ubicación se reiniciará la aplicación, pero tendrá que volver a iniciar la supervisión (*Figura 17).*

2.5. Pantalla

La pantalla es un gran pozo de consumo. A mayor pantalla del dispositivo, mayor consumo de batería. Por supuesto, si la aplicación se encuentra en primer plano y el usuario interactúa con ella, es responsabilidad del desarrollador que ésta limite el consumo en la media de lo posible sin entorpecer la Experiencia del Usuario (UX del inglés User eXperience).

Este problema de consumo se ha ido reduciendo a medida que ha evolucionado la tecnología utilizada para el *display* o pantalla del dispositivo. Algunas aplicaciones requieren de una iluminación determinada de la pantalla para su uso, aplicaciones como una linterna o una app que permita la lectura códigos QR o BiDi, aumentan al máximo su brillo para realizar una mejor función. Normalmente una aplicación no requiere modificar el nivel de luminosidad de la pantalla.

2.5.1. Buenas prácticas

Saber cuándo es necesario que el usuario visualice la pantalla en todo detalle y cuándo no es importante para poder gestionar esos cambios y limitar la luminosidad de la pantalla.
Detectar cuándo el dispositivo se encuentra en modo ahorro de batería también ayudará a optimizar el consumo de la pantalla.

```
    //1
self.sendNetworkOperation(params: ["Token":"Bearer aaaaaaaaaa","limit": 100],
                          completion: { (res) in
                                switch res.0{ //6
                                case true:
                                    //success
                                    break

                                case false:
                                    //error
                                    break
                                }
                           })

func isReachable() -> Bool {
    let reachability = Reachability()! //3

    if reachability.isReachable{ //4
        return true
    }
    return false
}

func sendNetworkOperation(params: Dictionary<String,Any>, completion: ((Bool,String?)->())){
    //2
    if self.isReachable(){
        //5
        //send operation
        completion(true, nil)

    }else{

        //5
        completion(false, "Error, network is not reachable")
    }
}
```

Figura 20: Ejemplo de accesibilidad de la red con Reachability.swift

En la *figura 18* se lanza un observador sobre `NotificationCenter`, que es el canal de notificaciones nativo de iOS. El observador lanzado permanece a la espera una notificación de cambio de estado de la batería, por ejemplo la activación del modo ahorro de batería. En ese momento lanza una llamada a `powerMode:` que debería actuar reduciendo el nivel de brillo de la pantalla entre otros cambios con el fin de respetar la acción de reducir el consumo.

Accediendo al `ProcessInfo` podemos obtener la información acerca del estado del modo ahorro para así realizar las acciones pertinentes (*Figura 19)*.

2.6. Red

La gestión inteligente de los accesos a la red permite que una aplicación pueda ayudar a la conservación de la batería. Si no hay conexión de red disponible, se deberían impedir los accesos a la red hasta que la conexión sea restablecida.

El principal problema ajeno al programador es la calidad de la señal que se recibe. Esto es debido a que con poca señal la transmisión es más lenta y amplía el tiempo de uso. También es conveniente evitar operaciones pesadas como *streaming* de vídeo. A no ser que se esté utilizando una conexión WiFi ya que este tipo de red consume mucha menos energía que las tecnologías 4G LTE, 3G… Esto es debido a que los dispositivos LTE utilizan servicios Multi-Entrada y Multi-Salida (MIMO del inglés Multiple-Input and Multiple-Output) la cual permite el uso concurrente de señales y permite mantener dos canales de conexión LTE simultáneamente. De manera similar, todas las antenas de red necesitan escanear en busca de señales de torres de repetición a las que poder conectarse.

```
class NetworkOperation{

    var isAPIServerReachable : Bool {
        get{
            return reachability.isReachable
        }
    }
    private var reachability : Reachability
    private var networkOperationQueue : OperationQueue = {
        let queue = OperationQueue()
        queue.maxConcurrentOperationCount = 1
        return queue

    }()

    init() {
        self.reachability = Reachability.init(hostname: “http://myapi.com/v1/“)!
        self.reachability.reachableOnWWAN = false

        NotificationCenter.default.addObserver(self, selector: #selector(networkStatusChanged),
name: ReachabilityChangedNotification, object: nil)

    }

    @objc
    func networkStatusChanged(){
        if self.isAPIServerReachable{
            self.networkOperationQueue.isSuspended = false
        } else{
            self.networkOperationQueue.isSuspended = true
        }
    }

    func performNetworkOperation(params: Dictionary<String,Any>, completion: @escaping ((Bool,
Any?)->())){
        self.enqueueRequest(params: params, completion: completion)
    }

    func enqueueRequest(params: Dictionary<String,Any>, completion: @escaping ((Bool, Any?)->())){
        let request = URL.init(string: "<<requestURL>>")!

        self.networkOperationQueue.addOperation({
            Alamofire.request(request, parameters: params).responseJSON{ response in
                switch response.result{
                case .success(let data):
                    completion(true, data)

                case .failure(let error):
                    completion(false, error)

                }
            }
        })
    }
}
```

Figura 21: Ejemplo monitorización de red y la ejecución de una cola de operaciones en red

Otro factor de este tipo que hace un uso intenso de estas tecnologías en concreto de WiFi, ocurre cuando esta se encuentra activada pero no está enlazada a ningún punto compatible, ya que realiza de manera continuada búsquedas de nodos WiFi a los que poder conectarse.
El uso de almacenamiento en memoria **caché*** evitará la descarga de componentes que se van a utilizar de manera continuada y no van a actualizarse necesariamente a corto plazo.

2.6.1. Buenas prácticas

Para terminar de conocer esta tecnología, se debería de tener en cuenta lo siguiente:

- Comprobar si hay una conexión de red apropiada antes de realizar cualquier operación de red.
- Monitorizar de manera continuada la disponibilidad de la red y actuar apropiadamente según su estado.

A continuación tenemos un ejemplo de código de cómo se podría comprobar de manera eficiente el estado de la red. Este ejemplo utiliza la librería Reachability.swift[6].
1. Se realiza una llamada al método encargado de realizar la operación de red, esta recibe como parámetro un diccionario con el contenido que se quiere enviar en la operación.
2. Para conocer el estado de la conexión de red se realiza una llamada al método `isReachable()`, este devolverá si es accesible o no.
3. Es creado un nuevo objeto del tipo `Reachability`.
4. Mediante el objeto que se ha creado en el paso 3, se accede a su propiedad `isReachable` que nos ofrecerá la información necesaria para saber si la red está accesible o no. Si es accesible el método devolverá un resultado true, en caso contrario el resultado será false.
5. Con el resultado de `isReachable` se enviará la operación o no. En ambos casos se devolverá un resultado a la llamada realizada en el paso 1.
6. El resultado recibido será la confirmación de que todo ha ido bien o hubo algún tipo de error, en caso de error se recibirá además un mensaje de error que se puede mostrar al usuario.

Para monitorizar la red y la ejecución de una cola de operaciones cuando la red esté disponible. Se ha utilizado las librerías Reachability.swift y Alamofire[7] (*Figura 21).*

- La clase `NetworkOperation` tiene una propiedad llamada `isAPIReachable` la cual es usada para comprobar si la red está accesible o no.
- Las propiedades `reachability` y `networkOperationQueue` son privadas, ya que no se quiere que estas puedan ser accesibles desde fuera
  - `reachability`. Nos ofrece información sobre la accesibilidad de la red.
  - `networkOperationQueue`. Es una cola de operaciones. Esta cola solo permitirá la ejecución de una operación al mismo tiempo.
- Del `NotificationCenter` obtenemos cuándo el estado de accesibilidad cambia y habilita o deshabita la cola de operaciones.
- El método `performNetworkOperation` genera y actualiza la lista de operaciones.
- Para crear y encolar las operaciones se utiliza el método `enqueueRequest`.

Las OperationQueue no pausan o suspenden la ejecución de ninguna operación. Una cola suspendida simplemente significa que la operación anterior no se ejecutará hasta que la actual termine su ejecución.

Usar peticiones de red basadas en colas es la mejor manera de no bombardear al servidor con multiples y simultáneas peticiones. Una solución plausible es utilizar dos colas distintas, una para la obtención de imágenes pesadas (datos no críticos para la aplicación) y otra con la que se obtendrá la información más necesaria.

También es una buena práctica encender o apagar esta cola de operación en función de la necesidad de la aplicación. Una cola con tareas que ya no se necesitan deberá ser cancelarlas, para evitar su futura ejecución.

2.7. Otro hardware

Cuando la aplicación está en segundo plano, se debe liberar el hardware que se esté utilizando:

- Cámara
- Altavoz, exceptuando las aplicaciones de música
- Micrófono

Las buenas prácticas para estas operaciones de hardware son las mismas: sólo deben ser

```
func shouldProceedWithMinLevel(with minLevel: Int) -> Bool{
    let device : UIDevice = UIDevice.current
    device.isBatteryMonitoringEnabled = true

    let state : UIDeviceBatteryState = device.batteryState
    if state == .charging || state == .full{
        return true
    }
    let batteryLevel : Int = Int(device.batteryLevel*100)
    if batteryLevel >= minLevel{
        return true
    }else{
        return false
    }
}
```

Figura 22: Conocer estado de la batería en Swift

inicializadas cuando la aplicación esté en primer plano y deben detenerse cuando vaya al segundo plano.

Las excepciones pueden ser el altavoz y o la conexión con un dispositivo Bluetooth. Si se va a desarrollar una aplicación de música, Bluetooth u otra aplicación relacionada con el audio, puede seguir utilizando el altavoz aunque la aplicación vaya a segundo plano. No se debe forzar la pantalla encendida para fines de audio. Del mismo modo, se debe continuar utilizando el Bluetooth si la aplicación tiene una transacción inacabada, como la transferencia de archivos con otro dispositivo.

Aplicaciones como Snapchat, la red social para compartir fotos de manera temporal con tus seguidores, dispone de un modo ahorro de batería. Estas son las funciones para ahorrar batería que utiliza:

- Desactivación de las notificaciones push, la vibración y notificaciones flash cuando se recibe alguna alerta.
- Reducción de la pre-carga de contenidos patrocinados (Vídeos).
- Mayor control del estado de la cámara, reduce su funcionamiento exclusivamente a la vista de captura de fotos y vídeo. Normalmente siempre está activa con el fin de ofrecer un acceso inmediato a la cámara.

2.8. Buenas prácticas generales

Una aplicación eficiente debe tener en cuenta el nivel de batería y su estado para determinar si realmente está haciendo una operación de uso intensivo de recursos. Otro punto a tener en cuenta es el estado de carga, si el dispositivo se está cargando o no.

Se debe utilizar la instancia `UIDevice` para recuperar el nivel de batería y su estado de carga. En la figura 22 se muestra el método `shouldProceedWithMinLevel:`, este toma un nivel mínimo de batería que se requiere para continuar con una operación dada que tiene la intención de realizar. El nivel es un número de coma flotante en el rango 0-100.

Existen algunas buenas prácticas para asegurar el uso prudente de la batería. Siguiendo estas buenas prácticas se podrían desarrollar aplicaciones eficientes desde el punto de vista energético:

- Minimizar el uso del hardware, iniciando la interacción con el hardware lo más tarde posible y detener esta interacción una vez que la tarea se haya completado.
- Comprobar el nivel de la batería y el estado de carga antes de iniciar tareas intensas.
- Si el nivel de la batería es bajo, hacer que el usuario decida si la tarea realmente se debe ejecutar.
- Intentar incluir una configuración para permitir que el usuario defina un nivel umbral de batería por debajo del cual la aplicación debe preguntar al usuario antes de ejecutar operaciones intensivas.

2.9. Nota Importante

Hay que medir el consumo energético de todos los dispositivos a los que está orientada la aplicación que se quiere desarrollar. Se deben identificar las áreas donde hay un consumo energético mayor y encontrar la mejor opción para reducirlo.

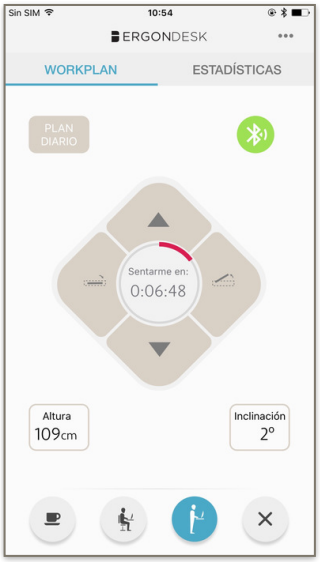

Figura 23: Vista de control remoto en iOS

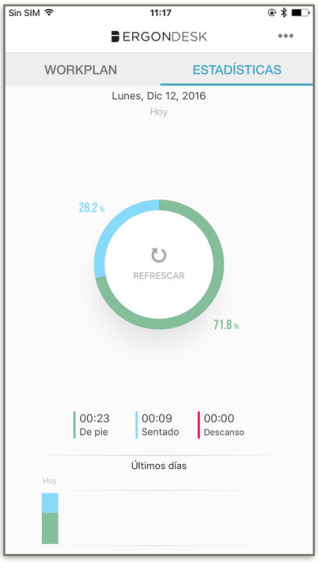

Figura 24: Vista de estadísticas del usuario en iOS

3. ErgonDesk

Idea original de PYNK SYSTEMS, presentada como proyecto de StartUp, premiada y con patrocinadores como Hyundai, AmCham EU… crearon un producto para lo que ellos llaman la oficina 2.0, mediante el cual se pudiera mejorar la integración y el trabajo en equipo. Con una mesa modular y configurable, quieren revolucionar el entorno de trabajo en la oficina.

ErgonDesk es una aplicación desarrollada para las plataformas móviles más utilizadas actualmente (iOS y Android). Con una cuidada interfaz, diseñada a partir de un estudio previo de usabilidad en aplicaciones móviles, permite el control de una mesa con servomotores la meta puede elevarse o inclinarse según las preferencias del usuario. Mediante una configuración previa de la aplicación con los datos de altura, peso y posibles problemas cervicales, esta genera un plan de trabajo con la mesa generando unos horarios de trabajo y descansos adecuados al usuario.

3.1. Funcionamiento de la aplicación

Una vez el usuario ha configurado la aplicación a su gusto, esta permite conectar con la mesa (ErgonDesk) mediante la tecnología Bluetooth, utilizada por su rango de alcance y por la fácil configuración que requiere por ambas partes: el teléfono móvil y la mesa.
Mediante una interfaz con forma de control remoto (como podría ser la de un mando de la televisión), se puede controlar la altura o inclinación de forma manual o mediante unos botones circulares en la parte inferior de esta. También se puede cambiar de manera automática a unas

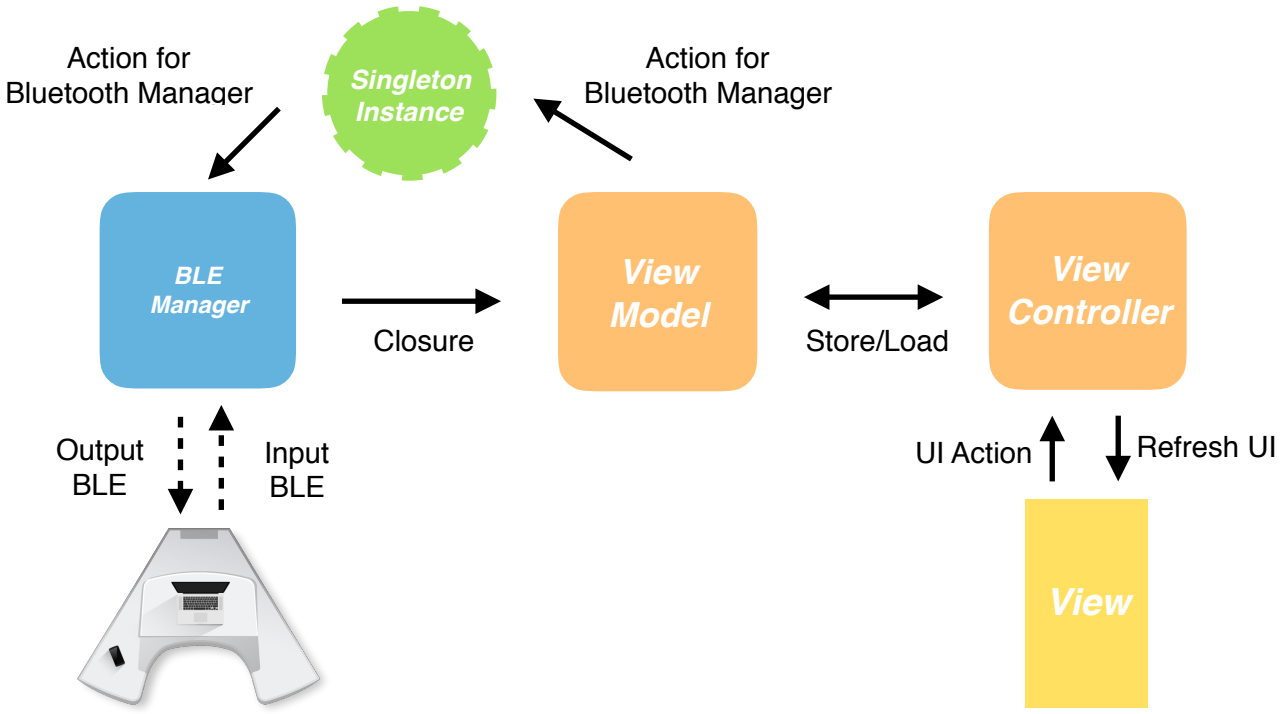

Figura 25: Arquitectura de código ErgonDesk

posiciones predeterminadas.

Cuando el usuario comienza la jornada (pulsando en el botón central del control remoto), este realiza una petición al servidor que realiza un cálculo con las preferencias del usuario y devuelve un plan diario que puede visualizarse en cualquier momento mediante el botón superior izquierdo.

Los cambios de posición entre sentado, de pie o descanso quedan registrados, para posteriormente mandar estos cambios al servidor que contiene el algoritmo que genera planes

```
static let sharedManager : BLEManager = BLEManager()

var discoveryClosure : ((CBPeripheral)-> (Void))?
var connectionEstablished : ((Bool)-> (Void))?
var errorConnectionClosure : ((Bool)-> (Void))?
var bleIsDownClosure : ((Bool)-> (Void))?

var debugMode : Bool = false
var centralManager : CBCentralManager!
var peripheral : CBPeripheral!
var bleServicesAreON : Bool = false
var services : [CBService]? = nil
var characteristics : [CBCharacteristic]? = nil

var desks : [Desk] = []
var peripheralName : String? = nil
var selectedDesk : Desk!
var actualPeripheral : CBPeripheral? = nil

var currentVC : UIViewController? = nil
lazy var alertsVC : AlertsController = AlertsController()

override init() {
    super.init()
    self.centralManager = CBCentralManager(delegate: self, queue: nil)
}
```

Figura 26: Variables contenidas dentro del manejador de Bluetooth

diarios personalizados. La transferencia de estos datos se realiza de manera segura.

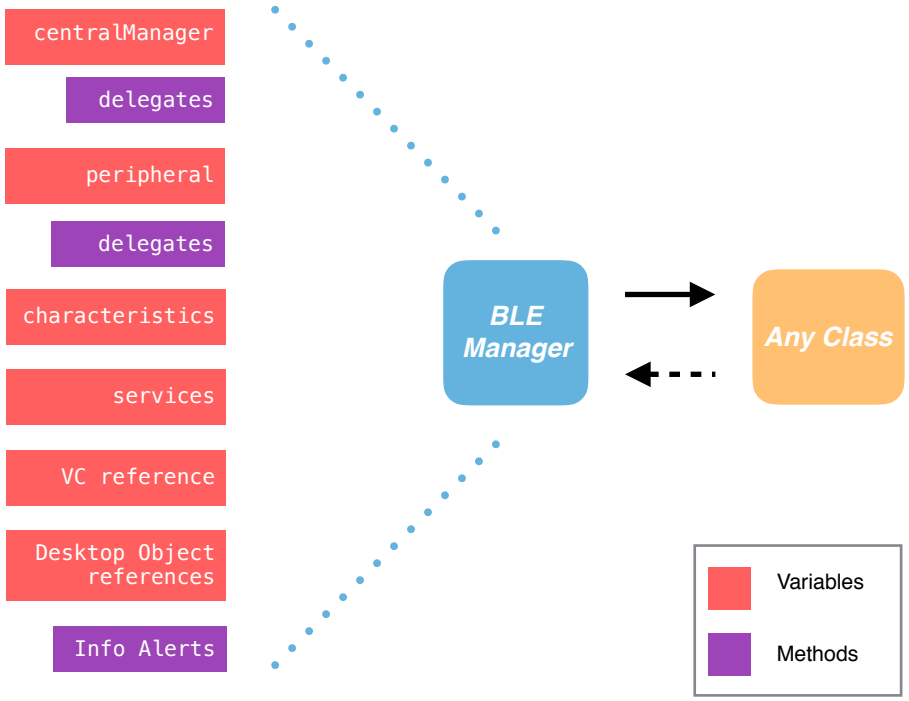

Figura 27: Arquitectura del manejador de Bluetooth

3.2. Arquitectura de la aplicación

Cuando se comenzó el desarrollo del proyecto, se decidió utilizar como modelo arquitectónico el patrón Modelo Vista Controlador (MVC del inglés Model View Controller), ya que resultaba más sencillo para el nivel de experiencia del pequeño equipo de desarrollo.

Para comprender este patrón, se explicará de manera sencilla cada una de las tres capas que lo componen:

- **Modelo:** es la capa encargada de almacenar y gestionar la información necesaria para el resto de capas, sólo puede comunicarse de manera directa con el Controlador.

- **Vista:** es la capa de interfaz que ve el usuario y con la que éste interactúa. Ésta envía las acciones únicamente al Controlador.

- **Controlador:** esta capa obtiene la información que le manda la Vista y la comunica al Modelo, para que éste realice las operaciones pertinentes.

A pesar de tener una estructura general muy clara y no demasiado compleja para futuros cambios, el uso que se realiza del manejador de Bluetooth (llamado BLE Manager dentro del proyecto) que podemos ver en la imagen 6, no es todo lo eficiente que debería ser, ya que no se contempla el manejo de posibles estados problemáticos que perjudican la correcta comunicación con la aplicación, y el motivo principal por el cual se ha utilizado como caso de estudio, el consumo energético.

El manejador de Bluetooth no solo se utiliza como tal, para manejar la comunicación del dispositivo y la ErgonDesk, también se utiliza como almacén de objetos del tipo *Desk*, a los cuales se acceden siempre que se necesite operar con una ErgonDesk. Para la comunicación con este *manager* utilizan **clausuras***, para devolver la información del estado en el que se encuentra. Esto es debido a que las señales vía Bluetooth se realizan de forma asíncrona, es decir que se ejecutan en otro hilo de ejecución distinto al principal, por lo que nunca se sabe cuándo sé obtendrá una resolución de la operación realizada (*Figura 26)*.

Para conocer en qué parte de la aplicación se encuentra el usuario, también se almacena en este manejador (BLE Manager), una referencia del controlador de la vista actual. Esta referencia del controlador es necesaria, ya que también se realiza el lanzamiento de alertas visuales para el usuario desde el mismo fichero que estamos describiendo.

Como se puede observar en la *figura 27*, existen varias lógicas distintas dentro del mismo fichero, lo que dificulta su lectura y entorpece al desarrollador en la adición de mejoras, teniendo que modificar dependencias con cada retoque realizado.

La falta de una organización mejor de las funciones del *BLE Manager* genera problemas de rendimiento debido al desconocimiento del estado de la tecnología en cada uno de los estados en los que se puede encontrar, dejando activas funcionalidades que no son necesarias en todo momento y perjudicando de manera grave al consumo energético.

3.3. Herramientas

Un correcto análisis del consumo energético necesita conocer tanto a nivel interno como a nivel externo qué información está ejecutando la aplicación y de qué manera. Para ello se han utilizado diversas aplicaciones de monitorización de procesos y análisis de datos.

3.3.1. Xcode

Principal entorno de desarrollo (IDE del inglés Integrated Development Environment) de

Figura 29: Ventana de creación de un proyecto de Xcode

aplicaciones para Mac, iPhone, iPad, Apple Watch y Apple TV.

Figura 28: XCode

3.3.1.1. ¿Qué es?

Xcode es un IDE de desarrollo para aplicaciones del entorno Apple. Para crear aplicaciones esta herramienta ofrece la posibilidad de crear interfaces de usuario, introducir código, realizar pruebas de software, realizar **depuración de código*** y publicar aplicaciones en la App Store (tienda de aplicaciones para Mac, iPhone, Apple Watch y Apple TV).

Este IDE puede descargarse de manera gratuita la versión más actual desde la App Store. Para obtener otras versiones de esta herramienta, es necesario disponer de una cuenta de desarrollador y acceder a la web de desarrolladores de Apple[8].

Xcode incluye una colección de compiladores de GNU (del inglés GNU is Not Unix), y puede compilar código C, C++, Swift, Objective-C, entre otros mediante una amplia gama de modelos de programación, como Carbón, Cocoa y Java.

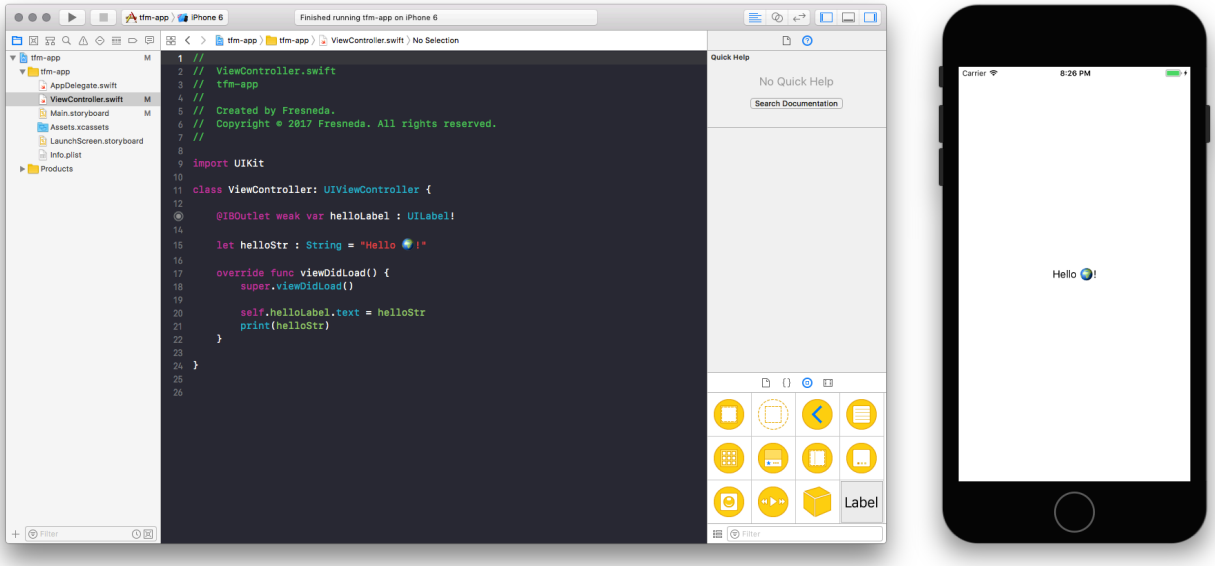

Figura 30: Xcode ejecutando una aplicación en Swift dentro del simulador

3.3.1.2. ¿Cómo funciona?

CREAR UN PROYECTO

Al iniciar el entorno de desarrollo por primera vez, este nos dará una serie de opciones para crear un proyecto:

- Crear un *Playground.* En la versión 6 de Xcode, se añadió esta opción como alternativa a crear un proyecto para probar código. Este tipo de proyecto nos permite generar sólo un fichero que se auto-compila y permite ejecutar código de manera sencilla y rápida. Sólo permite utilizar Swift como lenguaje de desarrollo. Un desarrollador polaco Krzysztof Zablocki[9] ha creado un plugin que permite utilizar *Playground* también en Objective-C.

- Crear un nuevo proyecto Xcode. Esta es la opción por defecto y la más habitual cuando se desea crear un proyecto nuevo. Al crear un proyecto de este tipo el IDE da la posibilidad de partir de una plantilla o empezar de cero. También da la opción de elegir el lenguaje de programación a utilizar (Objective-C ó Swift), utilizar o no una base de datos propia de la aplicación y la posibilidad de incluir **pruebas unitarias***.

- Clonar un proyecto de un repositorio. Si el proyecto está almacenado en un repositorio **GitHub***, esta opción permite clonarlo directamente a nuestro equipo para trabajar con él. No es necesario, pero se puede añadir una cuenta de GitHub para disponer más rápidamente de los proyecto propios alojados en este repositorio.

- Abrir un proyecto local existente. Permite abrir un proyecto ya existente en el ordenador. En la parte derecha mostrará los últimos proyectos abiertos con Xcode, en caso de querer abrir otro no visible en la lista se puede seleccionar mediante el explorador de ficheros.

DESARROLLAR Y EJECUTAR UN PROYECTO

Una vez creado y abierto un proyecto con Xcode podemos ver algo similar a lo que aparece en la imagen anterior. Consta de las siguientes secciones:

- Área de navegación del proyecto. Situado en la parte izquierda del IDE, muestra el árbol de directorios del proyecto. En su parte superior tiene una iconografía que indica las diferentes secciones que se pueden mostrar como búsquedas, errores, estado de los test o el visor de depuración de la aplicación.

- Editor de código. Por defecto en la zona central, es la sección que mayor espacio ocupa en la interfaz, permite la edición del código que va a ejecutarse en el proyecto. Este editor puede personalizarse con colores y distintos tamaños de fuente para mayor comodidad del usuario.

- Área de utilidades. Se comporta de manera diferente dependiendo qué esté haciendo el usuario. Si este está utilizando el editor de código el área de utilidades puede mostrar documentación de las clases, métodos y propiedades que está utilizando, también puede mostrar las propiedades que tiene el fichero que se encuentra abierto, dependiendo de la pestaña que tenga seleccionada de igual manera que el área de navegación. En la parte inferior aparece una sub-sección que muestra accesos rápidos para la adicción de contenido a la aplicación como elementos de interfaz, ejemplos de código, ficheros o contenido multimedia.

- Barra de herramientas. Situada en la parte superior de la aplicación, muestra información acerca del proyecto como el estado actual de la aplicación, el dispositivo sobre el cual se está

Figura 31: Visualización de la monitorización del consumo energético con Xcode

ejecutando la aplicación, y el esquema (schema en inglés) del proyecto. En la parte izquierda de esta barra unos botones permiten la ejecución/detención del proyecto. Los botones situados en la derecha permiten la visualización de cambios realizados en caso de utilizar un repositorio, la visualización de diferentes ficheros a la vez y la posibilidad de ocultar o mostrar partes de la interfaz.

- Simulador. Este es independiente y opcional en el desarrollo de aplicaciones en Xcode. Permite la simulación de un dispositivo real (con limitaciones) en el cual se pueden ejecutar los proyectos generados con Xcode. Para obtener el comportamiento real es recomendable utilizar un dispositivo real, el cual se puede seleccionar desde la barra de herramientas. Hasta Xcode 8.3 el dispositivo debía estar conectado por cable a la máquina que lanzaba la aplicación, con la versión 9.0 esto ya no es necesario y podemos lanzar ejecuciones de proyectos mediante una red WiFi.

Cuando se crea un proyecto nuevo con una plantilla `singleViewApplication` localizamos dos ficheros de código que permitirán la ejecución de un proyecto desde el primer momento: `AppDelegate.swift` y `ViewController.swift`.
`AppDelegate` es el fichero que se lanza en primera instancia al iniciar la aplicación, se encarga principalmente de los estados del ciclo de vida de la aplicación, cuando se inicia, se pausa o se finaliza.

`ViewController` es el controlador de una vista, que habitualmente va acompañado de un fichero de interfaz de usuario (*StoryBoard*). Este controlador de la vista contendrá la lógica, es decir, las acciones que recibirá la vista serán procesadas en este fichero. Dependiendo de la arquitectura de desarrollo a utilizar este fichero puede dividirse en varios ficheros (capas) con el fin de generar un código más limpio y permisivo con los futuros cambios.

El lenguaje que encontraremos en los ficheros antes mencionados dependerá del lenguaje seleccionado durante la creación del proyecto (Objective-C Swift). En proyectos creados con Swift es posible insertar código en Objective-C, como librerías. En Objective-C no es posible insertar Swift de forma nativa.

3.3.1.3. Su aplicación en el proyecto

Xcode ha sido el IDE principal de desarrollo durante todo el proyecto. También se ha utilizado para realizar las primeras comprobaciones de rendimiento de la aplicación gracias a su modo de monitorización de consumo.

3.3.2. Instruments

Instruments es una potente y flexible aplicación de análisis de rendimiento y pruebas incluida en las herramientas de Xcode.

Figura 32: Instruments

3.3.2.1. ¿Qué es?

Instruments es una aplicación diseñada con el fin de ayudar en el análisis de procesos y entender como funcionan y como optimizar las aplicaciones de macOS y iOS. Incorporando Instruments en el flujo de trabajo de aplicaciones, podemos ahorrar tiempo a la hora de detectar problemas futuros en el ciclo de desarrollo.

En Instruments, se utilizan herramientas especializadas conocidas como *instruments*, para trazar diferentes aspectos de las aplicaciones como procesos. Instruments recoge los datos acerca del perfil (consumo energético, consumo de CPU, consumo de red…) para poder realizar un análisis detallado a posteriori.

Al contrario que otras herramientas de depuración, Instruments permite obtener información de distintos perfiles de la ejecución de manera desacoplada. Esto hace más fácil identificar problemas que de otra manera podrían pasar inadvertidos.

3.3.2.2. ¿Cómo funciona?

Instruments se puede utilizar tanto en un dispositivo real como sobre el simulador de Xcode, pero

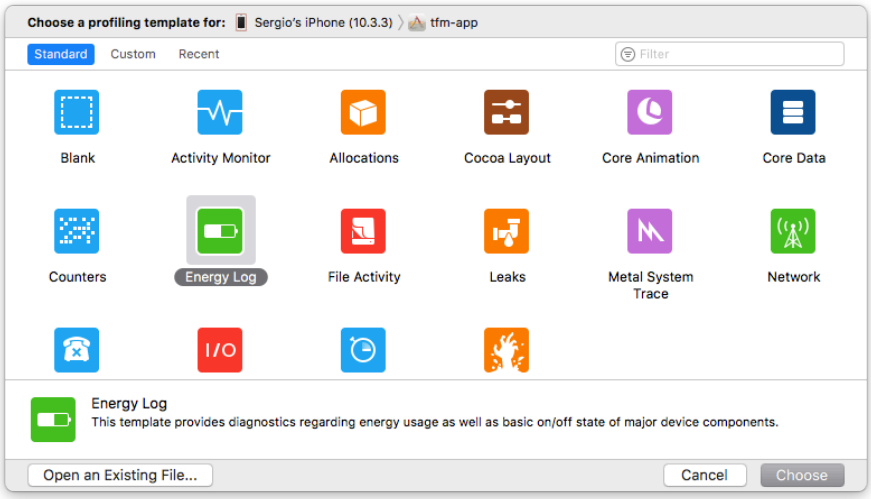

Figura 33: Ventana de selección perfiles en Instruments

para poder analizar el rendimiento de una aplicación de manera mucho más detallada es necesario utilizar un dispositivo real. Aunque Instruments viene dentro de Xcode, puede utilizarse de panera independiente.

Instruments permite realizar las siguientes acciones:
- Examinar cómo se comporta una o varias aplicaciones y sus procesos.
- Examinar características específicas del dispositivo, como el Wi-Fi y el Bluetooth.
- Analizar una aplicación en el simulador o en un dispositivo real.
- Crear trazas de *instruments* con el fin de analizar aspectos del comportamiento de la aplicación.
- Rastrear problemas en el código de la aplicación.
- Encontrar problemas de memoria en la aplicación, como pérdida de datos, secciones de memoria abandonadas y **zombies***.
- Identificar de qué manera se puede optimizar una aplicación para obtener una mejor eficiencia energética.
- Crear plantillas de instruments, para distintas aplicaciones.

Esta información ha sido obtenida de la web de desarrolladores de Apple[10].

3.3.2.3. Su aplicación en el proyecto

Instruments se ha utilizado durante el post-desarrollo del caso de estudio (en iOS) para detectar problemas de rendimiento y comprobar que las mejoras aplicadas realmente optimizan su consumo energético.

3.3.3. Charles Proxy

Charles es un **proxy*** que monitoriza los accesos a red que realiza un dispositivo, esto permite al desarrollador ver el tráfico HTTP (del inglés HyperText Transfer Protocol) y HTTPS (del inglés

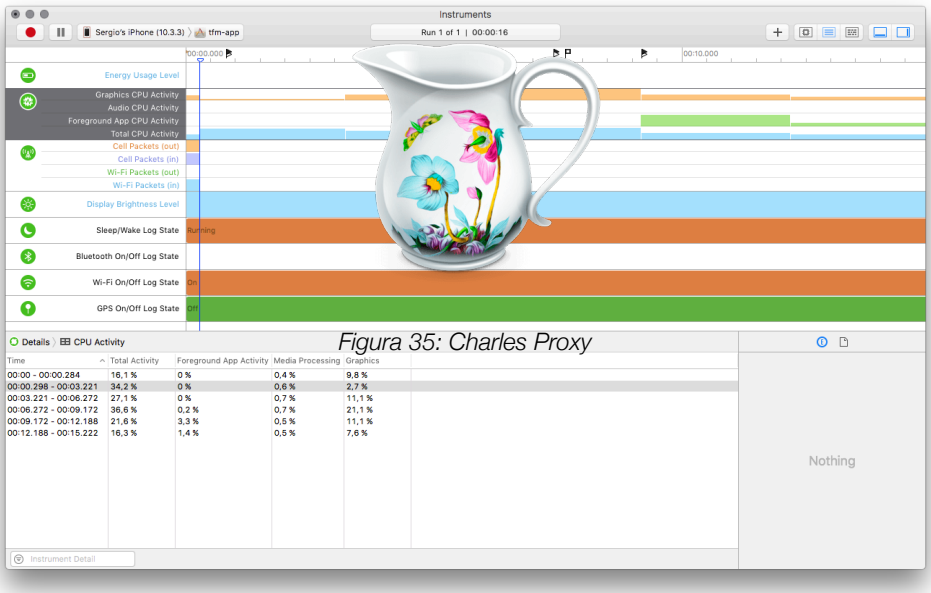

Figura 35: Charles Proxy

Figura 34: Visualización de instruments monitorizando el perfil de consumo energético

HyperText Transfer Protocol Secure) entre el dispositivo e internet. Esto incluye peticiones y respuesta; también las cabeceras HTTP que pueden contener **cookies*** y caché de información.

3.3.3.1. ¿Qué es?

Charles es una aplicación parcialmente gratuita que permite la creación de un proxy dentro de un ordenador, sin necesidad de alquilar o comprar un servidor para ello.

En el desarrollo Web, es difícil ver lo que se envía y recibe entre el navegador web (cliente) y el servidor. Sin esta visibilidad, es difícil y requiere mucho tiempo determinar exactamente dónde está el problema. Charles hace que sea fácil ver lo que está sucediendo, para que se pueda diagnosticar y solucionar problemas

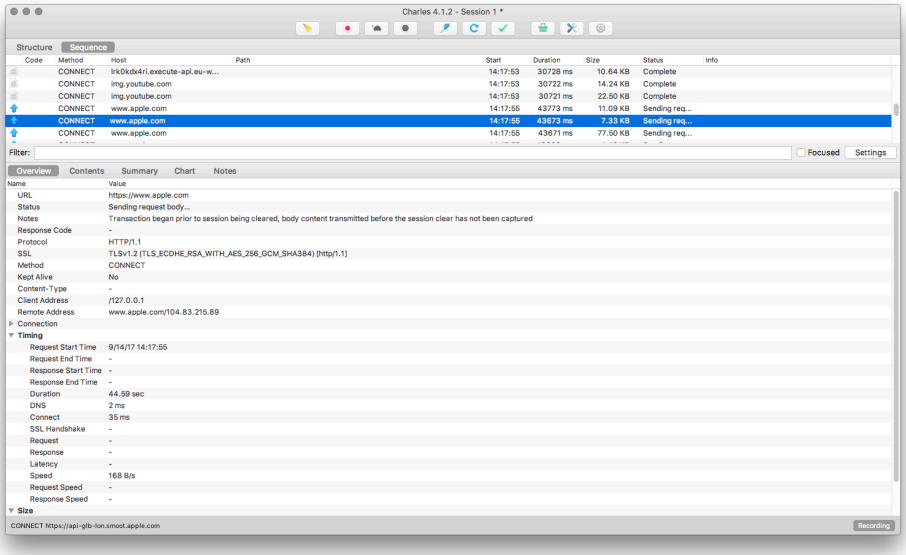

Figura 36: Visualización de peticiones de red con Charles Proxy

rápidamente.

Durante su ejecución sin licencia se dispone de 30 minutos de ejecución de la aplicación. Una vez superado este tiempo, la aplicación se detendrá. En caso de utilizar esta aplicación sin la compra de una licencia, ésta no limitará el uso de ninguna funcionalidad.

3.3.3.2. ¿Cómo funciona?

Una vez obtenida la aplicación Charles desde su página web e iniciado el proxy se configuran los dispositivos que van a realizar accesos a red a través de Charles. Una vez realizada la configuración, toda la información de red será almacenada para un posterior análisis.

Estas son las funcionalidades clave que muestran en la página web de Charles Proxy[11]:
- SSL Proxying. Posibilidad de ver peticiones **cifradas*** mediante el protocolo SSL (del inglés Secure Socket Layer).
- Limitación del ancho de la red. Limitar el tamaño de la red con el fin de simular conexiones lentas o con mucha **lantencia***.
- Depuración de AJAX. Facilitar la lectura de peticiones XML y JSON en forma de árbol o como texto.
- AMF. El contenido **Flash*** y **Flex*** puede ser analizado en forma de árbol.
- Repetir peticiones de red con el fin de probar los cambios que se realizan en el servidor.
- Editar peticiones de red.
- Añadir o editar puntos de ruptura en las peticiones o respuestas.
- Validar respuestas HTML, CSS (del inglés Cascading Style Sheets) y RSS (del inglés Rich Site Summary) utilizando una validación **W3C***.

3.3.3.3. Su aplicación en el proyecto

Charles Proxy ha sido una herramienta complementaria a Instruments, que permite el análisis de los datos recibidos y enviados mediante red y la comprobación de los parámetros y la respuesta generada en la aplicación.
Al contrario que Instruments, esta herramienta ha sido utilizada tanto durante su desarrollo como durante su post-desarrollo.

4. Monitorización de ErgonDesk

A continuación se detalla cada uno de los casos de uso y tras un análisis de consumo, se comenta cuáles son los problemas de consumo energético que se detectaron, la explicación de porqué se ha seleccionado como problema y cual sería la mejor manera de solventarlos.

4.1. Casos de Uso

Para una mejor monitorización de la aplicación es necesario separar esta en Casos de Uso (CU, en inglés UC del inglés User Cases). Con esto se acota los posibles problemas de una forma clara.

4.1.1. CU 001: Búsqueda de ErgonDesk

En este momento, el dispositivo comprueba que el Bluetooth está activado y comienza a buscar una *ErgonDesk*. Una vez encuentra un dispositivo hace una primera conexión a él, para obtener los servicios de éste. En caso de ser válidos, almacena el identificador del dispositivo y refresca la interfaz de usuario para mostrarle la disponibilidad del dispositivo detectado.
Cuando el usuario interactúa sobre la celda de la *ErgonDesk* detectada, muestra una alerta de confirmación para realizar la conexión. Si esta alerta es aceptada, realiza la conexión y avisa al usuario mediante un mensaje dependiendo del resultado exitoso o fallido de la conexión (*Figura 37).*

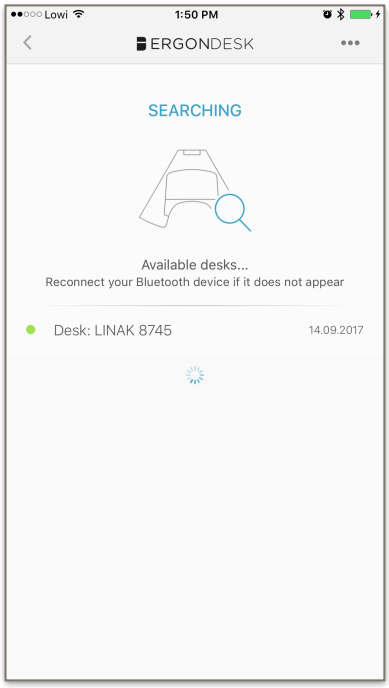

Figura 37: Búsqueda de ErgonDesk

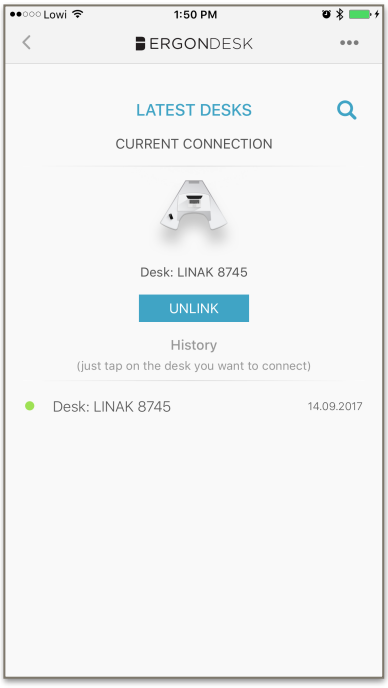

Figura 38: Reconexión de ErgonDesk

4.1.2. CU 002: Reconexión de ErgonDesk

Una vez la aplicación se ha conectado con al menos un dispositivo con anterioridad, este CU muestra por pantalla un listado con todos las *ErgonDesk* a las que se ha conectado en el pasado. Cada una de las entradas de la lista muestra un indicador gris o verde, dependiendo si esta se encuentra en el rango de la señal de Bluetooth. Al seleccionar una de las celdas de la lista disponibles con el indicador verde, se procede a realizar la conexión con la *ErgonDesk*. Si la conexión resulta exitosa aparece un mensaje de confirmación en la pantalla. Finalmente si se realizó el emparejamiento la vista se cierra y manda al usuario a la vista de control remoto de la mesa (*Figura 38).*

4.1.3. CU 003: Control remoto

Este es el CU principal de la aplicación. Desde este se puede controlar de manera manual o automática la altura e inclinación de la *ErgonDesk*. Una botonera central simula los controles de un mando a distancia, con el cual se puede mover de manera manual centímetro a centímetro, la altura y la inclinación. En su parte inferior dispone de cuatro botones, para realizar el desplazamiento o detención de la *ErgonDesk* de manera automática, respetando el rango definido por las preferencias del usuario. Cada vez que se pulsa uno de estos botones se envían una serie de llamadas a la mesa que modifican el valor de sus características, lo que posteriormente interpreta la caja de control de la mesa y mueve los motores. Cuando una característica es modificada, se lanza otra señal a la mesa para que informe acerca de su altura e inclinación actual, para mostrarlo al usuario. Desde la parte superior podemos ir a otros dos CU: el plan diario y la búsqueda de mesas.

Si se pulsa el botón central, la aplicación realizará una conexión al servidor para obtener el plan diario del usuario, y lo iniciará de manera inmediata.
Si en algún momento se pierde la conexión con la *ErgonDesk,* la aplicación mostrará un mensaje informando al usuario y ofreciéndole la posibilidad de realizar una reconexión (*Figura 39)*.

Figura 39: Control remoto

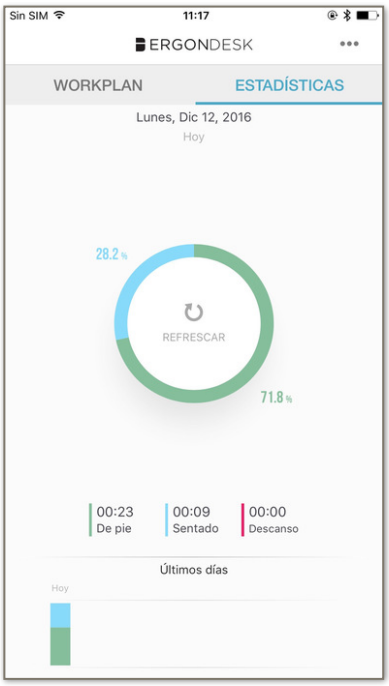

Figura 40: Estadísticas

4.1.4. CU 004: Estadísticas

Este CU nos informa mediante un gráfico circular del porcentaje de tiempo que el usuario ha estado en cada uno de los estados de su jornada laboral (sentado, de pie, descanso). En la parte central de este gráfico circular hay un botón, también con forma redondeada, que permite el refresco manual de los datos del gráfico. En la parte inferior aparece otro gráfico, éste en forma de barras apiladas, el cual muestra la misma información que el anterior pero de los últimos siete días.
Los datos de ambos gráficos se obtienen durante la primera presentación de la vista. Al mostrar esta pantalla por primera vez, se lanza una petición al servidor, el cual devuelve los datos que deben pintar los gráficos. En caso de pulsar el botón de obtención manual de los datos, se realiza la misma petición. Esta puede realizarse cuantas veces se pulse el botón (*Figura 40)*.

4.2. Problemas

Búsquedas ineficiente de nuevos dispositivos
Cuando se realizan las búsquedas para conectar con un dispositivo no se realiza un filtrado previo, es decir cada vez que la aplicación detecta un dispositivo (lo haya detectado anteriormente o no) prueba si los servicios de que dispone son los que debe de tener la ErgonDesk. Esta incidencia acarrea tiempos de búsqueda mayores y mayor consumo de CPU, ya que se comprueban los servicios de todos los dispositivos que se encuentran para después descartarlos o admitirlos (*Figura 41*).

```
//MARK: - CBCentralPeripheralDelegate
// Invoked when you discover the peripheral's available services.
func peripheral(peripheral: CBPeripheral, didDiscoverServices error: NSError?) {

    var isErgonDesk = false


    if peripheral.services != nil{
        //check each peripheral services
        for service in peripheral.services!{
            if ErgonDeskManager.validService(service){
                isErgonDesk = true
               //if is a valid service try to discover his characteristics
                peripheral.discoverCharacteristics(nil, for: service)
            }
        }
        if isErgonDesk{
            print("\n ERGONDESK Discovered | \(peripheral.identifier.uuidString)\n\n")
        }
    }
}
```

Figura 41: Código problemático ErgonDesk BLE Manager, detección de nuevos periféricos compatibles

Este problema se encuentra descrito en el caso de uso de búsqueda de una ErgonDesk como en el de reconexión con ErgonDesk.

Control de los estados de la búsqueda
No existen llamadas de inicio o detención de la búsqueda dentro del manejador de Bluetooth. Estas llamadas se realizan directamente desde el controlador externo. Esta mala práctica elimina la posibilidad de controlar de manera real el estado de escaneo de dispositivos, permitiendo que en ocasiones se mantengan búsquedas en casos de uso que no están destinados a realizar esta tarea (*Figura 42)*.

```
    override func viewDidAppear(animated: Bool) {
        super.viewDidAppear(animated)

        //Start to scan new devices
        self.bleManager.centralManager.scanForPeripheralsWithServices(nil, options: nil)

        //When BLE discover a new ERGONDEK device send a closure
        bleManager.discoveryClosure = { peripheral in
            self.searchTableView.reloadData()
        }
        self.bleManager.currentVC = self
    }
    override func viewDidDisappear(animated: Bool) {
        super.viewDidDisappear(animated)
        //Stop scan
        self.bleManager.centralManager.stopScan()
    }
```

Figura 42: Código problemático ErgonDesk BLE Manager, inicio y detención del escaneo

Este problema se da en el caso de uso de búsqueda de ErgonDesk y en el de reconexión con ErgonDesk. También es posible que suceda en el control remoto.

Búsquedas ineficiente de dispositivos conocidos

Al realizar una búsqueda sobre los dispositivos con los que ya se estableció una conexión con anterioridad, se realiza una recuperación de los dispositivos almacenados en disco, posteriormente se compara de forma manual cada periférico dentro el rango con las ErgonDesk a los recuperadas de disco. La comparación se realiza mediante el UUID (*Figura 43)*.

```
        let idDesktop = BLEManager.sharedManager.desks[indexPath.row].idDesktop
        let filtered = BLEManager.sharedManager.availablePeripherals.filter{
                $0.identifier.UUIDString == idDesktop
        }
        if filtered.count > 0 {
                BLEManager.sharedManager.desks[indexPath.row].isAvailable = true
        }
        else{
                BLEManager.sharedManager.desks[indexPath.row].isAvailable = false
        }
```

Figura 43: Código problemático ErgonDesk BLE Manager, filtrado de periféricos conocidos

```
    var timerBluetooth : NSTimer!

    //Init the timer with 0.2 seconds of delay
    self.timerBluetooth = NSTimer.scheduledTimerWithTimeInterval(0.2, target: self, selector:
#selector(modifyTilt), userInfo: nil, repeats: true)

    //MARK: - Move desktop Methods
    //Adapter for move up the desktop from selector
    func moveDesktopTOP(){
        self.moveDesktop(.TOP)
    }
    //Adapter for move down the desktop from selector
    func moveDesktopBOTTOM(){
        self.moveDesktop(.BOTTOM)
    }
    //Method for move up the desktop
    func moveDesktop(direction: typeMovement){
        if self.bleManager.peripheral != nil {

            //Send signal to ErgonDesk for writte a new value
            BluetoothProvider.modifyDesktopValues(self.bleManager.peripheral, characteristic:
self.btSendCharacteristics![0], movement: direction)
            if !self.model.guestMode && self.model.programs.count != 0 &&
self.model.timerBluetooth != nil {
                //Send signal to ErgonDesk for read the current Value
                let height = (BluetoothProvider.readDesktopHeight(self.bleManager.peripheral,
characteristic: self.btReadCharacteristics![0])/100)+Constants().legsHeight
                    .
                    .
                    .
            }
        } else {
            //Reset timer
            if self.timerBluetooth != nil {
                self.timerBluetooth.invalidate()
            }
        }
```

Figura 44: Código problemático ErgonDesk BLE Manager, envío de señales

Envío de señales

Al pulsar cualquiera de los botones del control manual, se lanza una señal de modificación de altura o inclinación a la mesa. En el caso de mantener pulsado el botón se inicia un temporizador (*timer* en inglés) que lanza señales hacia la *ErgonDesk*, de manera continuada cada 2

milisegundos. El uso de estos timers sobrecarga el procesador, haciendo que este ejecute muchas comprobaciones de manera continua. Una de cada tres señales que se envían a la mesa no son procesadas, debido a que ésta solo asegura la correcta lectura de señales en periodos de 3 milisegundos. Por este motivo se está desperdiciando energía en envíos innecesarios (*Figura 44)*.

Lectura de la mesa

Para obtener la altura e inclinación se realizan accesos de lectura a ciertas características habilitadas por el dispositivo remoto que ofrecen esta información. Cada lectura se realiza en el proceso de conexión, iniciando así el contador de la interfaz del usuario. También se leen estos datos durante el proceso de envío de señales que modifican la altura y la inclinación, haciendo uso como mínimo de dos señales por parte de la aplicación en cada modificación que se realiza (*Figura 44)*.

Petición de datos de manera innecesaria

Como se mencionaba en la explicación del caso de uso del control remoto, cada vez que es pulsado el botón de refresco manual de los datos estadísticos, la aplicación manda una señal al servidor para obtener los nuevos datos sin realizar ningún tipo de comprobación de cuándo se recibieron los últimos datos. Cada petición no solo implica la acción de mandar una señal vía red y la respuesta resultando, sino también la inicialización de los componentes necesarios y el re-dibujado de la vista (*Figura 45)*.

```
@IBAction func refreshCircleChart(sender: AnyObject) {
    //Reset charts values
    self.statisticsModel.todayData?.breakTime = 0
    self.statisticsModel.todayData?.standingTime = 0
    self.statisticsModel.todayData?.sittingTime = 0

    self.loadDataFromBackEnd()
}

func loadDataFromBackEnd(){
    //If user have token try to retrieve the charts data
    if UserDefaultsManager.loadTokenUser() != ""{
        StatisticsProvider.getStatistics(){
            (result: Int, statistics: [Statistic]?, today: Statistic?) in
            switch result {
            //The retrieve is success set the new data in to the charts
            case 0:
                if statistics != nil{
                    self.statisticsModel.dataUserStatistics = statistics!
                }
                if today != nil{
                    self.statisticsModel.todayData = nil
                    if self.statisticsModel.todayData == nil{
                        self.statisticsModel.todayData = today
                    }else{
                        self.statisticsModel.todayData?.breakTime += (today?.breakTime)!
                        self.statisticsModel.todayData?.standingTime += (today?.standingTime)!
                        self.statisticsModel.todayData?.sittingTime += (today?.sittingTime)!
                    }
                }
                break
            //The retrieve is failure do not change nothing
            case 1:
                print("Error loading Statistics")
                break
            default:
                print("¿Error loading Statistics?")
                break
            }
            //Refresh Charts
            self.loadDataFromLocal()
        }
    }
}
```

Figura 45: Código problemático ErgonDesk BLE Manager, petición de datos a red

Dibujado de gráficos
El dibujado de gráficos suele implicar el uso de librerías externas, muchas veces de gran tamaño. Los cálculos que se realizan son costosos y deben ser realizados solo cuando realmente sean necesarios (*Figura 45)*.

4.3. Soluciones

Búsquedas ineficientes de nuevos dispositivos
Cuando se conoce el tipo de dispositivo a buscar, se pueden realizar búsquedas de dispositivos directamente filtrando por los identificadores únicos de sus servicios. El método `ed_findDevices:` es el encargado de iniciar la búsqueda, este método recibe dos parámetros opcionales `services:` y `options:`. Cuando se inicia la búsqueda, se cancela cualquier búsqueda anterior que estuviese iniciada `ed_stopScan`, posteriormente se comprueba que no hay otras instancias de búsqueda iniciadas con `ed_isScanning`. Finalmente se inicia una nueva búsqueda y se crea una operación `addCancelScanOperationForInnactive` para cancelar la búsqueda en caso de que no se encuentren dispositivos en un periodo de tiempo (*Figura 46)*.

```
    /// Stop scan and try to find devices with specific services
    ///
    /// - Parameters:
    ///   - services: Array with CBUUIDs
    ///   - options: Array of String:Any
    func ed_findDevices(with services: [CBUUID]? = nil, options:  [String:Any]? = nil){
        if k_DebugBLE{ print("Searching BLE Devices") }

        // Stop the posible current scan
        self.ed_stopScan()

        // Check if CentralManager is scan
        if !self.ed_isScanning(){
            // Scan peripherals with a certain services
            self.centralManager.scanForPeripherals(withServices: services, options: options)
            // Set a new operation for stop the scan if they spent more than 30 seconds without
              detect a new device
            self.addCancelScanOperationForInnactive()
        }
    }
    func centralManager(_ central: CBCentralManager, didDiscover peripheral: CBPeripheral,
advertisementData: [String : Any], rssi RSSI: NSNumber) {
        // Check if the new detected peripheral is on the list
        guard !self.peripheralList.contains(peripheral) else {
            // break the execution of this method
            return
        }
        // Add a new operation for cancel the scan with a delay of 30 seconds
        self.addCancelScanOperationForInnactive()

        // Add the new peripheral to the list
        self.peripheralList.append(peripheral)
        }
```

Figura 46: Código solución ErgonDesk BLE Manager, detección de nuevos periféricos compatibles

Control de estado de búsqueda
Mediante la implementación de métodos que controlen e informen del estado actual. Centralizar el control en estos métodos impide que se creen varias instancias de búsqueda que dificulten la posterior detención. Cuando se implementan los delegados de la librería `CoreBluetooth` es importante que se apoye en estos métodos. En el código podemos ver el método delegado `centralManager:didConnect:` que fuerza la detención de la búsqueda `ed_stopScan()`(en caso de que se esté realizando la búsqueda), una vez se ha emparejado correctamente con un dispositivo (*Figura 47)*.

```
    /// Check and provide info about the active peripheral
    /// States:
    /// disconnected, connecting, connected, disconnecting
    ///
    /// - Returns: Enum
    func ed_deskStatus() -> CBPeripheralState {
        return self.actualPeripheral?.state ?? CBPeripheralState.disconnected
    }

    /// Check and provide if scanning state is ON in CBManager
    ///
    /// - Returns: Boolean
    func ed_isScanning() -> Bool{
        return self.centralManager.isScanning
    }

    /// Provide privately the current state of CBManager
    /// States:
    /// unknown, resetting, unsupported, unauthorized, poweredOff, poweredOn
    ///
    /// - Returns: Enum
    func ed_bluetoothStatus() -> CBManagerState{
        return self.centralManager.state
    }

    /// If CBManager is scanning, force a stop
    func ed_stopScan() {
        if self.ed_isScanning(){
            self.centralManager.stopScan()
        }
    }
    /// Invoked when a connection is successfully created with a peripheral.
    ///
    /// - Parameters:
    ///   - central: BLE Manager
    ///   - peripheral: Peripheral
    func centralManager(_ central: CBCentralManager, didConnect peripheral: CBPeripheral) {
        if k_DebugBLE{  print("\n Connected | On \(peripheral) \n") }

        peripheral.delegate = self
        peripheral.discoverServices(k_ServicesCollection)

        self.ed_stopScan()

        //Alert to caller, the current device has been connected.
    }
```

Figura 47: Código solución ErgonDesk BLE Manager, métodos de control de estados

Búsquedas ineficientes de dispositivos conocidos

Buscando dispositivos cuyos UUID únicos son conocidos la problemática se reduce enormemente. No es necesario buscar todos los dispositivos disponibles en el rango de la aplicación, si no que se pueden lanzar instancias de conexión a los dispositivos conocidos y devolver el resultado de este intento de conexión, para conocer su disponibilidad (*Figura 48)*.

```
    /// Allow connect with a BLE devices.
    ///
    /// - Parameter peripheral: Peripheral Object
    func ed_connect(with peripheralIdentifier: UUID, services: [CBUUID]? = nil, options:
[String:Any]? = nil) {
        guard self.ed_deskStatus() != .connecting && self.ed_deskStatus() != .connected else{
            if k_DebugBLE { print("refusing to connect \(self.ed_deskStatus().rawValue)") }
            return
        }
        //Returns a list of known peripherals by their identifiers.
        self.actualPeripheral = self.centralManager!.retrievePeripherals(withIdentifiers:
[peripheralIdentifier]).first
       .
       .
```

Figura 48: Código solución ErgonDesk BLE Manager, búsqueda de dispositivos conocidos

Envío de señales

Una solución que ofrece un ahorro significativo en el consumo de energía por parte de la aplicación es la eliminación de timers. Utilizando el manejador de gestos nativos (`UIGestureRecognizer` en iOS), se puede realizar la misma acción siempre y cuando se controlen los estados de éste. Con esto se permite el envío de señales cada periodo de tiempo que sea necesario, en este caso cada 3 mili segundos y un control total de cuando se debe detener el movimiento de la mesa (*Figura 49)*.

```
    longPressGesture.minimumPressDuration = 0.3

    @IBAction func longPressAction(_ sender: UILongPressGestureRecognizer){
        //Retrieve the button pressed
        if let button = self.buttonCollection.filter({$0.isTouchInside}).first{

            switch sender.state {
            //The gesture recognizer has received touch objects recognized as a continuous
gesture.
            case .began:
                self.controllerButtonPressed(button)
                break
            //The gesture recognizer has received touches resulting in the cancellation of a
continuous gesture.
            case .cancelled:
                self.controllerButtonPressed(self.stopSignal)
                break
            //The gesture recognizer has received touches recognized as the end of a continuous
gesture.
            case .ended:
                self.controllerButtonPressed(self.stopSignal)
                break

            default:
                break
            }
        }
    }
    func controllerButtonPressed(_ sender: UIButton){
        //Sends the corresponding signal
        switch sender{
        case self.goUpButton:
            self.movver_tellViewModel(event: EDRemoteControlActions.VCtoVM.moveDeskUp)
        case self.goDownButton:
            self.movver_tellViewModel(event: EDRemoteControlActions.VCtoVM.moveDeskDown)
        case self.lessTilt:
            self.movver_tellViewModel(event: EDRemoteControlActions.VCtoVM.moveDeskLessTilt)
        case self.moreTilt:
            self.movver_tellViewModel(event: EDRemoteControlActions.VCtoVM.moveDeskMoreTilt)
        case self.stopSignal:
            self.movver_tellViewModel(event: EDRemoteControlActions.VCtoVM.stopSignals)
        default:
            assert(true, "Unexpected button")
        }
    }
```

Figura 49: Código solución ErgonDesk BLE Manager, envío de señales

Lectura de la mesa

Mediante una opción de subscripción a las características de los servicios del periférico, podemos obtener de manera gratuita la altura e inclinación de la *ErgonDesk*. Esto es debido a que cada vez que se realiza una modificación sobre la mesa, ésta realiza una devolución con todos los valores de sus características. Así la aplicación solo tiene que filtrar qué valor necesita para mostrarlo al usuario (*Figura 50).*

Petición de datos de manera innecesaria

Conociendo el periodo de tiempo en el cual pueden aparecer datos nuevos, se deberían controlar las peticiones que se realizan al servidor, en caso de que se obtengan los mismos datos que los

que ya se tienen almacenados, desechar los nuevos y cancelar el refresco de la vista del usuario (*Figura 51)*.

```
    /// Subscribe or Unsubscribe to all services available from peripheral received in params
    ///
    /// - Parameter peripheral: CBPeripheral
    /// - subscribe: Boolean value
    private func ed_subscribeToServices(from peripheral: CBPeripheral, subscribe: Bool){
        //Check if peripheral have services

        guard let services = peripheral.services else{
            if k_DebugBLE { print("Peripheral \(peripheral.identifier.uuidString) dont have
services available") }

            return
        }

        // Subscribe in all characteristic of BLE device
        // Service  -> Characteristic 1
        //          -> Characteristic 2
        //         ...
        //          -> Characteristic n

        for service in services{
            if let characteristics = service.characteristics{
                for characteristic in characteristics{
                    if k_DebugBLE { print("Subscribing in \(subscribe), to characteristics \
(characteristic.uuid.uuidString)") }
                    if characteristic.isNotifying{
                        peripheral.setNotifyValue(subscribe, for: characteristic)
                    }
                }
            }
        }
```

Figura 50: Código solución ErgonDesk BLE Manager, lectura de la mesa0

```
    func getFutureTimeStamp() -> NSDate{
        //Get the current date with 5 minutes more
        let calendarNow = NSCalendar.currentCalendar()
        return calendarNow.dateByAddingUnit(.Minute, value: 5, toDate: NSDate(), options: [])!
    }

    @IBAction func refreshCircleChart(sender: AnyObject) {
        //Allows to obtain information only every 5 minutes,
        //this is the time the data is updated on the server
        guard let lastStatisticDate = self.statisticsModel.dataUserStatistics?.first?.date where
lastStatisticDate.timeIntervalSince1970 > self.getFutureTimeStamp().timeIntervalSince1970  else{
            return
        }
        self.loadDataFromBackEnd()
        self.statisticsModel.todayData?.breakTime = 0
        self.statisticsModel.todayData?.standingTime = 0
        self.statisticsModel.todayData?.sittingTime = 0
    }
```

Figura 51: Código solución ErgonDesk BLE Manager, petición de datos de manera inmediata

Dibujado de gráficos

La realización de cálculos menos costosos o el uso de librerías más ligeras, orientadas a una función en lugar de tener multiples posibilidades y solo utilizar una pequeña parte.

4.4. Cambios tras la aplicación de las mejoras

Los siguientes datos se obtuvieron antes y después de la aplicación de los cambios para la mejora del consumo energético. El número de pruebas realizadas en ambos casos fue de al menos 50 veces por caso de uso. Se utilizaron dos ErgonDesk para las pruebas. El entorno de pruebas elegido fue una oficina de trabajo, esto fue debido a que es el entorno habitual para el que está diseñado la ErgonDesk. El número medio de dispositivos Bluetooth en el entorno de pruebas era de 10; entre ordenadores portátiles, de sobremesa y teléfonos móviles de múltiples fabricantes.

Los dispositivos móviles que se utilizaron fueron:

- iPhone 6S+ (iOS 10.3.2) - (iOS 10.3.3)
- iPhone 6S (iOS 9.0)
- iPhone SE (iOS 9.2.1)

NOTA: A la siguiente sección aparecerá el término impacto energético (Energy Impact). Este término es referido a la siguiente información obtenida de la web de desarrolladores de Apple[12].

> El impacto energético es la información proporcionada en tiempo real sobre el uso de la energía de una aplicación mientras se ejecuta, esta información es plasmada en un gráfico (*Figura 31*) de la actividad relacionada con la energía consumida recientemente. Las áreas siguientes están representadas por el indicador de la tabla de impacto energético:
>
> - Costes generales: Las barras azules (*Figura 31*) ilustran la energía que la propia aplicación utiliza para realizar su trabajo (entre un 1% y un 30%). Las barras rojas muestran la energía adicional que utilizan los recursos del sistema que deben estar encendidos para realizar el trabajo de la aplicación(desde 31% hasta más de un 100%) […]
>
> La información recopilada en todas estas áreas se utiliza para presentar una clasificación de energía para la aplicación. Cuando el usuario interactúa con la aplicación, el impacto energético debe ser bajo a menos que el usuario haya decidido iniciar una operación intensiva. Cuando el usuario no está interactuando con la aplicación, no debería haber ningún consumo energético.

4.4.1. Búsqueda

Una vez aplicadas las mejoras en la búsqueda de dispositivos se detectó una substancial mejora en los tiempos de respuesta. Debido a la aplicación de filtros en las búsquedas y a la reducción del tratamiento que se realizaba a cada nuevo dispositivo encontrado, el tiempo de respuesta se redujo en más de un 60%.

Como se puede observar en la *tabla 2*, el tiempo de búsqueda de la primera versión al realizar esta tarea de forma completa tiene una media de 25,3 segundos en detectar. Sin embargo, tras la mejora este tiempo se redujo hasta unos 10,5 segundos de media. Esto es debido a que las búsquedas se realizan con conocimiento de qué se pretende encontrar, es decir utilizando las opciones que ofrece la API `CoreBluetooh` para realizar un escaneo más concreto obviando aquellos dispositivos que no cumplen con las características necesarias.

Al comprobar el consumo energético, se detectó que en ambas versiones de la aplicación el consumo al detectar un nuevo dispositivo se elevaba al menos al 100%, la explicación es la carga que supone al sistema la conexión y desconexión del periférico al detectarlo. Quizás este problema deba a la no total optimización de la librería `CoreBluetooth` por parte de Apple.

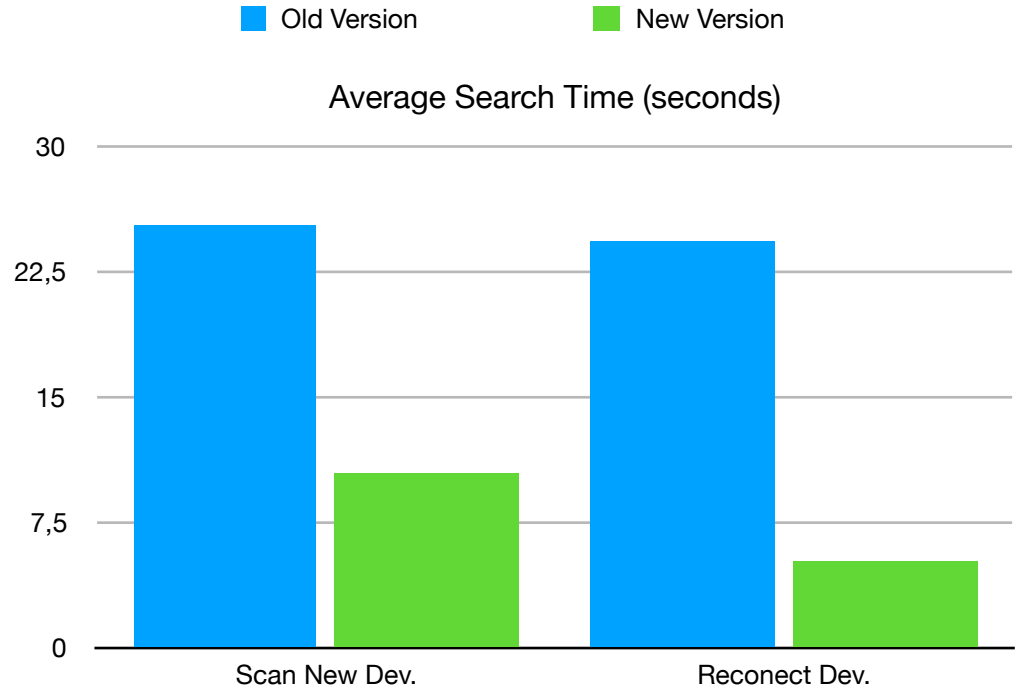

Tabla 2: Tiempo de operación medio en la búsqueda de dispositivos

Otro factor a destacar acerca del consumo energético es el consumo mínimo o constante, aquel que se mantiene durante la mayoría del tiempo (*Tabla 3*). Mientras que en la primera versión el consumo mínimo oscilaba entre un 30% y un 35% considerado como un impacto energético alto (según la monitorización de consumo de Xcode), en la nueva versión se reducía este valor hasta un 10% considerado como un consumo normal o bajo. La explicación de este fenómeno es debido a dos razones; el control de instancias de búsqueda de dispositivos activas y el filtrado de los dispositivos por servicios.

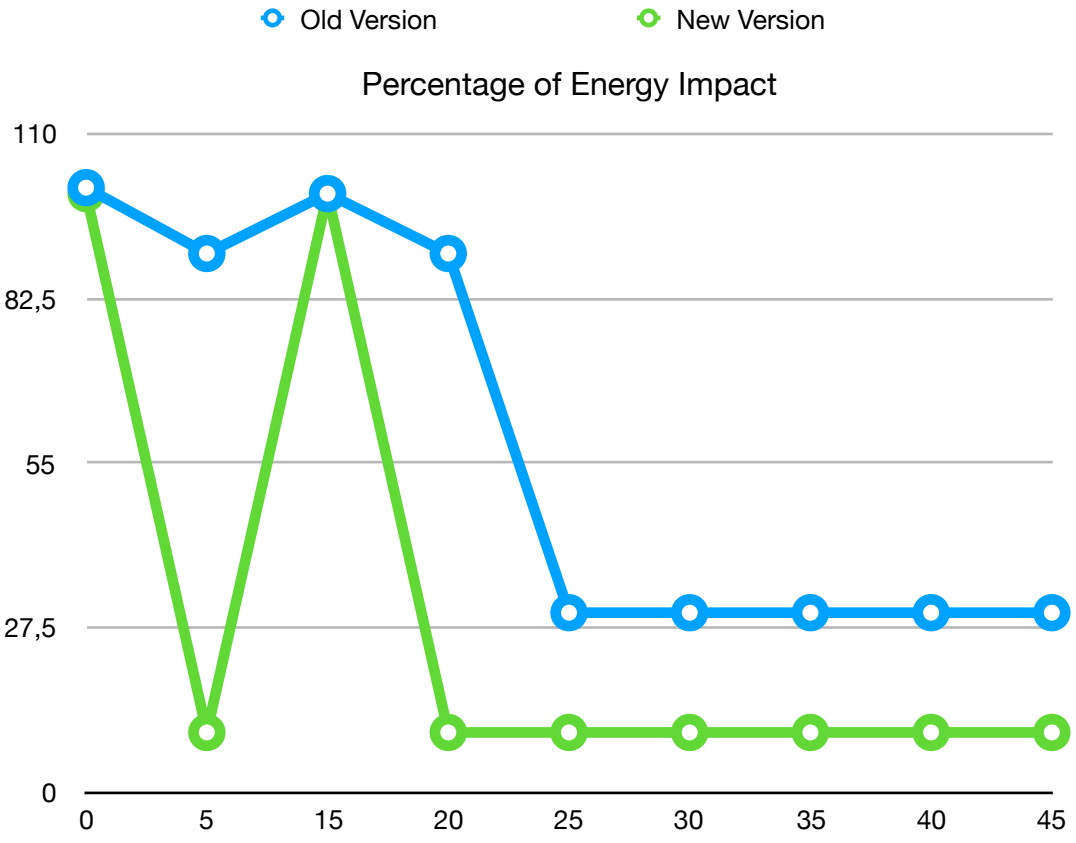

Tabla 3: Impacto energético en la búsqueda de dispositivos en un periodo de 45 segundos

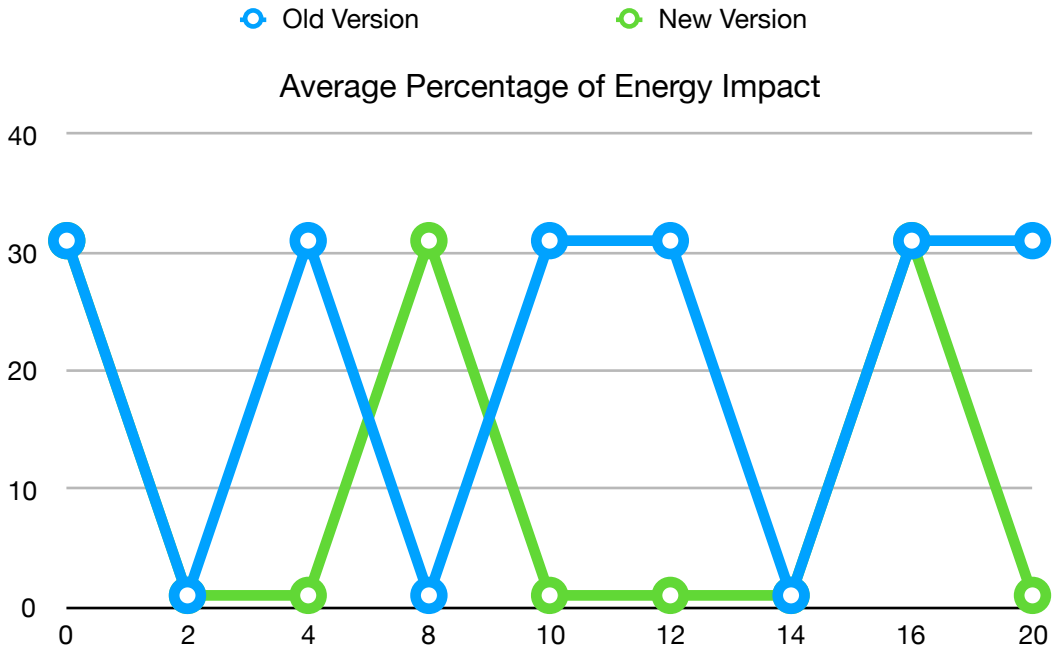

Tabla 4: Impacto energético envío y recepción de señales

Una vez aplicada la mejora de código de envío y recepción de señales hacia la ErgonDesk se observa no sólo un menor consumo energético sino también una mayor precisión cuando se modifican los valores de altura de la mesa, es decir; cuando se fija la altura a la que debe de estar la mesa y comienza a moverse esta se fija con un error de milímetros.

En la *tabla 4* se aprecia en ambas versiones una variación aparentemente continua de valores, esto es debido al envío y a la recepción de señales. En la versión sin mejoras los valores son mayores debido al uso de timers y a la lectura manual de los valores de la mesa. En cambio con la versión energéticamente eficiente, existen variaciones pero muy inferiores a la primera versión ya que no tiene dependencia de realizar una lectura manual cada vez que se modifica un valor en la

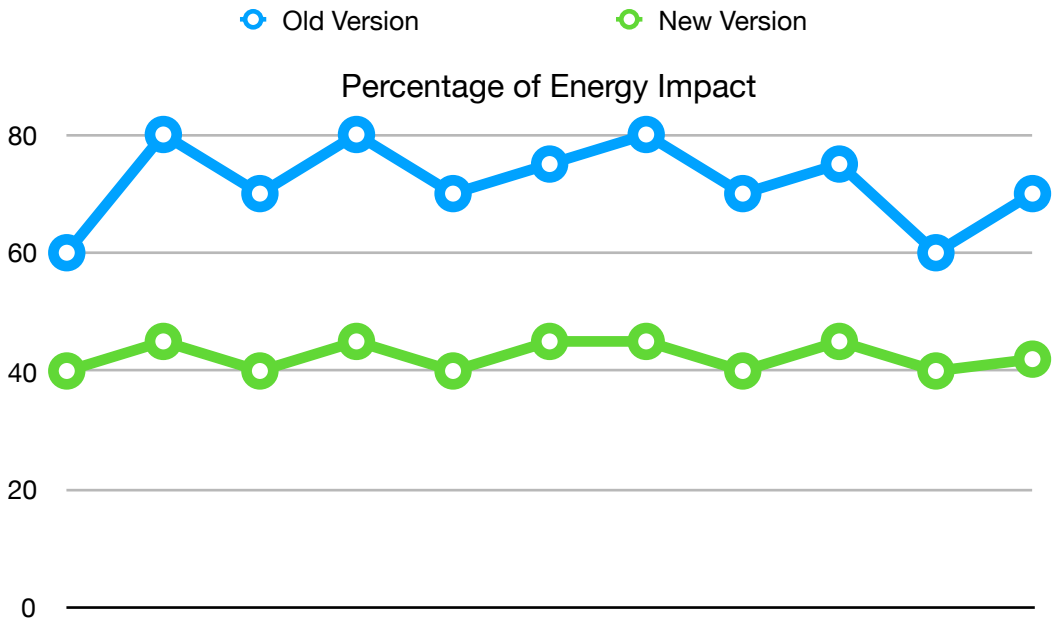

Tabla 5: Impacto energético medio por minuto en la consulta de datos en la sección de estadísticas y dibujado de la vista

mesa, sino que es ésta la que devuelve el valor de forma automática gracias a la subscripción de las características.

Cuando se solicitan datos de las estadísticas del usuario se realiza una descarga de información del servidor y posteriormente se dibuja la vista mediante una animación. Esta solicitud se puede realizar de dos formas: de manera manual pulsando en el botón central de la pantalla o al entrar en la misma. Debido a las mejoras aplicadas ya no se realizan consultas innecesarias por lo que se evita todo el proceso anterior, solo se podrán obtener datos cuando realmente puedan aportar nueva información al usuario.

En la *tabla 5* se observa qué consumo medio ha realizado esta vista sobre la aplicación en un periodo de 20 minutos.

Durante la realización de este proyecto de mejora energética, se han realizado en el código las siguientes modificaciones:

MANEJADOR BLUETOOTH
- Número total de líneas en el proyecto original: 244
- Métodos totales en el proyecto original: 16

- Número total de líneas en el proyecto tras la mejora: 472
- Métodos totales en el proyecto tras la mejora: 25

CONTROL REMOTO
- Número total de líneas en el proyecto original: 1883
- Métodos totales en el proyecto original: 73

- Número total de líneas en el proyecto tras la mejora: 1705
- Métodos totales en el proyecto tras la mejora: 61

ESTADÍSTICAS
- Número total de líneas en el proyecto original: 518
- Métodos totales en el proyecto original: 17

- Número total de líneas en el proyecto tras la mejora: 451
- Métodos totales en el proyecto tras la mejora: 14

El proyecto sobre el que se realizaron estas modificaciones tiene más de 10.000 líneas de código y más de 10 casos de uso diferentes. El proyecto inicial se desarrolló en un periodo de 4 meses.

5. Conclusión

Llegamos al final de la memoria del proyecto, en el que hablaremos de las conclusiones que hemos sacado tras finalizar la mejora de la aplicación y de todo el proceso que ha conllevado.

5.1. Resultado del proyecto

Tras varios meses de trabajo, el objetivo principal era conseguir desarrollar una mejora de la aplicación que respetara de la manera más eficientemente posible la batería del teléfono. Esto ha sido una ardua tarea, pero finalmente se pudo completar.

El código inicial tenía una longitud aproximadamente de unas 11.300 líneas de código, de las cuales no se consiguieron reducir prácticamente, pero el fin era modificar la aplicación para obtener una mejora energética sin interferir en la arquitectura, sólo utilizando buenas prácticas.

La mayor motivación después de completar el proyecto ha sido ver como estas mejoras que he aplicado en este proyecto han sido revisadas por el equipo técnico de mi actual empleo, y serán tomadas como referencia en los siguientes proyectos tanto de iOS como de manera más teórica pero igualmente aplicable en Android.

5.2. Todo lo aprendido

Para completar el proyecto del que trata esta memoria he tenido que leer y repasar gran cantidad de documentación sobre las distintas tecnologías que intervienen en el consumo energético en dispositivos móviles. Mi experiencia como desarrollador de iOS, antes de trabajar en un proyecto de mejora, ha sido una difícil aventura en algunos momentos de ésta, pero completamente satisfactoria una vez finalizada. He tenido que adquirir y expandir conocimientos para entender mejor el funcionamiento de una aplicación móvil.
La investigación sobre las tecnologías utilizadas para cada una de las partes del proyecto me ha ayudado a tener más claro cómo funcionan y tomar ideas para futuros proyectos propios.
Podría decir que, además de un Proyecto Final de Master, esto ha sido una gran experiencia para mí, que me resultará muy útil para mi futuro profesional. Aplicando los conocimientos que explico en esta memoria, me doy cuenta de que ahora soy mejor desarrollador.

5.3. Opinión personal

Mi valoración en general es muy positiva en todos los sentidos. Todo esto me ha ayudado a crecer como persona y como programador, lo que me enorgullece enormemente.
Después de haber completado el trabajo, me doy cuenta de todo el camino que he recorrido a pasos agigantados. Tantas horas de duro trabajo, han asentando las bases para continuar desarrollando aplicaciones eficientes. Todo esto ha sido una gran experiencia que no podré olvidar.
Este proyecto me ha servido para poder darme cuenta del interés que despierta en mí el mundo de los dispositivos móviles. Tengo claro que quiero enfocar mi carrera profesional al diseño de software móvil.

6. Glosario

Glosario de términos técnicos ordenados alfabéticamente.

** Según Wikipedia[13], una **API** (del inglés Application Programming Interface) es un conjunto de subrutinas, funciones y procedimientos que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.*

** Según Wikipedia[14], el tipo de dato **Booleano** es en computación aquel que puede representar valores de lógica binaria, esto es 2 valores, valores que normalmente representan falso o verdadero.*

* ***Caché*** son datos almacenados por un componente físico o de software para un futuro uso, permitiendo una ejecución más rápida por parte del sistema ya que estos datos los tiene almacenados y no es necesario obtenerlos de nuevo.

** Según Wikipedia [15], El **cifrado** es un procedimiento que utiliza un algoritmo con cierta clave para transformar un mensaje, sin atender a su estructura lingüística o significado, de tal forma que sea incomprensible o, al menos, difícil de comprender a toda persona que no tenga dicha clave.*

** Según Wikipedia[16], una **clausura** es una función evaluada en un entorno que contiene una o más variables dependientes de otro entorno. Cuando es llamada, la función puede acceder a estas variables. El uso explícito de clausuras se asocia con la programación funcional.*

** Según Wikipedia[17], una **cookie**, es una pequeña información enviada por un sitio web y almacenada en el navegador del usuario, de manera que el sitio web puede consultar la actividad previa del usuario.*

** Un **delegado** es un objeto que actúa en nombre de, o en coordinación con, otro objeto cuando ese objeto encuentra un evento en un programa.[18]*

** Según Wikipedia[19], la **depuración de código** es el proceso de identificar y corregir errores de programación. En inglés se conoce como debugging, porque se asemeja a la eliminación de bichos (bugs), manera en que se conoce informalmente a los errores de programación.*

** **Flash** es un contenido animado normalmente creado por el software llamado Adobe Flash.*

** **Flex** es un conjunto de ficheros creados con el software Adobe Flex.*

** Según Wikipedia[20], un **framework** es una abstracción en la cual el software que proporciona funcionalidad genérica puede ser cambiado selectivamente por código adicional escrito por el usuario.*

** **GitHub** es un sitio web que permite alojar proyectos utilizando el sistema de control de versión Git.*

** Según Wikipedia[21], una **GPU** (del inglés Graphics Processor Unit) es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos o aplicaciones 3D interactivas.*

** Una **notificación push** es un mensaje que utiliza habitualmente un dispositivo móvil. Los desarrolladores de aplicaciones pueden enviarlos en cualquier momento para avisar al usuario; Los usuarios no necesitan tener la aplicación ejecutándose o estar utilizando sus dispositivos para recibir estas notificaciones.*

** Según Wikipedia[22], un **proxy** o servidor proxy, en una red informática, es un servidor —programa o dispositivo—, que hace de intermediario en las peticiones de recursos que realiza un cliente (A) a otro servidor (C).*

** Una **prueba unitaria** es una forma de comprobar que una porción o unidad del código tiene un funcionamiento correcto.*

* ***Zombies*** de memoria *es un término utilizado para problemas de código que se mantienen en ejecución consumiendo recursos y no son controlados por ningún proceso.*

** **W3C**, es el principal consorcio de estándares de la World Wide Web.*

7. Referencias

Lugares de internet en los que he podido encontrar la información útil que me ha ayudado a desarrollar este proyecto.

[1] Apple (01/08/2017), Performance Tips — Recuperado de: https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/PerformanceTips/PerformanceTips.html

[2] GeekBench (01/08/2017), iOS Benchmarks — Recuperado de: https://browser.geekbench.com/ios-benchmark

[3] Raywenderlich GitHub (01/08/2017), swift-algorithm-club — Recuperado de: https://github.com/raywenderlich/swift-algorithm-club

[4] Varish, Gaurav (2016). High Performance iOS Apps. — Recuperado de: http://shop.oreilly.com/product/0636920034506.do

[5] Raywenderlich GitHub (01/08/2017), QuickSort — Recuperado de: https://github.com/raywenderlich/swift-algorithm-club/blob/master/Quicksort/Quicksort.swift

[6] Ashleymills GitHub (01/08/2017), Reachability.swift — Recuperado de: https://github.com/ashleymills/Reachability.swift

[7] Alamofire GitHub (01/08/2017), Alamofire — Recuperado de: https://github.com/Alamofire/Alamofire

[8] Apple (01/08/2017), Apple Developer — Recuperado de: https://developer.apple.com

[9] Zabłocki, Krzysztof (01/08/2017), Playgrounds — Recuperado de: https://github.com/krzysztofzablocki/Playgrounds

[10] Apple (01/08/2017), Instruments User Guide — Recuperado de: https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/index.html

[11] Charles Proxy (01/08/2017), About Charles — Recuperado de: https://www.charlesproxy.com/overview/

[12] Apple (1/08/2017), Measure Energy Impact with Xcode — Recuperado de: https://developer.apple.com/library/content/documentation/Performance/Conceptual/EnergyGuide-iOS/MonitorEnergyWithXcode.html

[13] Wikipedia (01/08/2017). Interfaz de programación de aplicaciones — Recuperado de: https://es.m.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones

[14] Wikipedia (01/08/2017). Tipo de dato lógico — Recuperado de: https://es.m.wikipedia.org/wiki/Tipo_de_dato_l%C3%B3gico

[15] Wikipedia (01/08/2017). Cifrado (criptografía) — Recuperado de: https://es.m.wikipedia.org/wiki/Cifrado_(criptograf%C3%ADa)

[16] Wikipedia (01/08/2017). Clausura (informática) — Recuperado de: https://es.m.wikipedia.org/wiki/Clausura_(inform%C3%A1tica)

[17] Wikipedia (01/08/2017). Cookie (informática) — Recuperado de: https://es.m.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)

[18] Apple (01/08/2017). Delegation — Recuperado de: https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html

[19] Wikipedia (01/08/2017). Depuración de programas — Recuperado de: https://es.m.wikipedia.org/wiki/Depuraci%C3%B3n_de_programas

[20] Wikipedia (01/08/2017). Framework — Recuperado de: https://es.m.wikipedia.org/wiki/Framework

[21] Wikipedia (01/08/2017). Unidad de procesamiento gráfico — Recuperado de: https://es.m.wikipedia.org/wiki/Unidad_de_procesamiento_gr%C3%A1fico

[22] Wikipedia (01/08/2017). Servidor proxy — Recuperado de: https://es.m.wikipedia.org/wiki/Servidor_proxy