Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Effects on performance and usability for cross-platform application development using React Native

by

Niclas Hansson, Tomas Vidhall

LIU-IDA/LITH-EX-A–16/043–SE

June 16, 2016

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköpings universitet
Institutionen för datavetenskap

Final thesis

Effects on performance and usability for cross-platform application development using React Native

by

Niclas Hansson, Tomas Vidhall

LIU-IDA/LITH-EX-A–16/043–SE

June 16, 2016

Supervisor: Anders Fröberg

Examiner: Erik Berglund

Abstract

A big problem with mobile application development is that the mobile market is divided amongst several platforms. Because of this, development time gets longer, more development skills are needed and the application gets harder to maintain. A solution to this is cross-platform development, which allows you to develop an application for several platforms at the same time. Since September 2015 the cross-platform framework React Native, created by Facebook, has been available for public use. This thesis evaluates React Native, for both Android and iOS, in regards to performance, platform code sharing as well as look and feel. An application was developed for both platforms, one version using the native language and one version using React Native. The different versions were compared through automated test scenarios to evaluate performance, manual code review for platform code sharing and with a user study to evaluate the look and feel. The results show promise as the user study shows that the React Native versions of the application have similar user experiences as their native counterparts without significantly affecting performance. The results also show that for the specified application about 75% of the React Native code could be used for both platforms, while it was easy to add platform-specific code.

Contents

Chapter 1

Introduction

Mobile application development has become a big area within the software development industry. Meanwhile the user base is spread out over mobile phones that use different operating systems. To cover a large extent of the market you have to develop the application for more than one platform. Developing an application in more than one language is time consuming and therefore also costly. This has exerted the need for developing applications targeting many OS's at once. Instead of writing several applications there are so called cross-platform development techniques which allows for the development of an application that works on more than one OS. There are many frameworks that use existing web technologies to create apps that work on many platforms. However, depending on the application, it can be hard to achieve a native feeling when actual native components are not used. Facebook has developed a new framework, called React Native, which uses native scripting to create actual native components. It allows development of mobile applications, using concepts derived from the web framework ReactJS, and create them in a similar way to how web applications are developed using ReactJS. Since it creates actual native components it needs some platform-specific code, but it is possible to share a significant amount of code between platforms.

1.1 Motivation

The purpose of this thesis is to evaluate Facebook's new framework React Native. The React Native framework promises the concept "Learn once, write everywhere" which means that once the developer has learned the React Native framework he or she will be able to apply it everywhere, i.e. to multiple platforms. It also means that building native-specific applications is required, however when the same framework is used for all applications, structures and code can be reused which greatly decreases the time consumption of developing subsequent applications after the first one. Further-

more, developers familiar with the ReactJS web framework should be able to quickly start developing Android and iOS applications without prior knowledge in the native languages Java and Swift respectively. This is because of the similarities between the React Native framework and the ReactJS web framework. Instead of having the need for competence within Web, Android and iOS development a company could reduce this to only needing React developers, which could be valuable. [1, 2, 3]

Research into the effects on application development has been done for cross-platform frameworks such as Xamarin and PhoneGap. The result showed that in some instances cross-platform development can have negative influences on performance metrics such as CPU and memory usage [4], the look and feel of the application[5], or both. Therefore, a new cross-platform framework such as React Native needs to be evaluated to determine if the advantages of the framework make up for potential drawbacks in performance and in the look and feel of the developed application. When using React Native it is important to be aware of its advantages and disadvantages to be able to decide if the framework will suit the development needs of the intended application.

1.1.1 Attentec

The work performed in this thesis was conducted at Attentec, a consulting firm that specializes in software development and IT solutions. Attentec works a lot with web technologies as well as with mobile platforms and aim to deliver bleeding edge technological solutions to their customers. Therefore, they are always looking into the potential of new technologies and frameworks that will keep them at the forefront of industry.

Attentec can benefit from React Native if the applications produced by the framework meet their quality requirements. A cross-platform framework can be used to lower the development time and consequently the cost for cross-platform mobile applications. With a reduced development cost Attentec would be able to offer their customers even more affordable cross-platform applications which could secure business that would otherwise not have happened. Achieving a good look and feel for an application is important as that greatly affects the user experience and therefore it is the most time consuming phase in Attentec's development process. For a cross-platform framework to truly be valuable it needs to be able to deliver a look and feel that is equal to a native Android or iOS application.

1.2 Research questions

The focus of this report is to answer the questions pertained in this section.

1. Is the performance of a React Native application better or worse than the same application developed in a native language?

2. How much of the codebase written using React Native can be used for both iOS and Android?

3. Can an application created using React Native achieve the same look and feel as a native application?

1.3 Aim

The goal of this thesis is to evaluate the cross-platform framework React Native. Furthermore, the aim is to develop an application using React Native and develop the same application in the native languages for Android and iOS in order to be able to compare the React Native app to their native counterparts.

1.4 Delimitations

Due to the platform support of React Native the only investigated platforms are Android and iOS. Other platforms, such as Windows Phone are therefore not included.

Chapter 2

Theory

This chapter will present the underlying theory of the work, development techniques and evaluation techniques. The chapter will first describe some different cross-platform techniques and how React Native works in section 2.1. This is followed by a description of native Android and iOS development in section 2.2 and section 2.3. Thereafter, techniques that can be used to evaluate a mobile application are described in section 2.4. This chapter also contains other related work that was found when researching the topic of this report, this can be found in section 2.5.

2.1 Cross-platform development

In the world of mobile cross-platform development there have been many different ways to develop applications with different advantages and disadvantages. This report presents mobile web applications, hybrid applications, cross-compiled applications and native scripting applications. React Native uses the native scripting approach.

2.1.1 Mobile Web applications

A mobile web application is actually not a native application which can be downloaded and installed, but rather a regular web application that has been adapted to the mobile format. It is run inside the browser on the mobile and behaves like a regular web application. As every mobile that runs a browser can run the application, it is cross-platform compatible. However, it will be restricted to the use of native components and gestures that are implemented in the browser. The application is developed using regular web techniques such as HTML, CSS and JavaScript. One advantage for mobile web applications is that the application will never need to be updated or even installed on the mobile device, since it is hosted on a server. Another advantage is that it will have the same look and feel on different

devices. However, the application has to support different resolutions and screen sizes, since one single application should work on all devices. It will also be unavailable to an offline user, since an Internet connection is required to access it. Many large websites, such as YouTube and Facebook, are accessible through a regular webpage as well as through a mobile web application, which is developed with smaller display sizes in mind. [5, 6]

2.1.2 Hybrid applications

A hybrid application is developed using a webview-component while also having access to native APIs. It will result in a mobile application that has to be installed on the device. The application will be restricted like a mobile web application since most of it is built using a webview, which is essentially a native component that encapsulates a browser. The application is built using web techniques, with the ability to call native APIs using a JavaScript hardware abstraction layer to get access to camera, GPS and other hardware components in the device. Hybrid applications can reuse interface components, like a mobile web application, however it is hard to achieve an actual native feeling. Further drawbacks from using a web browser as core component in hybrid applications includes lower performance compared to native applications. Some examples of hybrid development frameworks are PhoneGap, Trigger[1] and Ionic[2]. [4, 5, 7]

2.1.3 Cross-compiled applications

A cross-compiled application is an application written in a non-native language which can be compiled into a fully native application using a cross-compiler. The entire application is developed using a cross-compiler framework, for example Xamarin using C#, which will then be able to compile the correct native binaries for different platforms. This will result in a native application that will have to be installed on a device. Since the code is compiled into platform-specific files real native components can be used and therefore achieve a true native feeling in the application. However, since each platform is different, specialized code for each platform is required and the applications will not be able to share 100% of the codebase. If an application uses a lot of platform-specific features, the platform-specific version of the application could essentially become like two entirely separate applications. This approach is highly dependent on the efficiency of the chosen cross-compile framework. [4, 5]

2.1.4 Native scripting applications

Native scripting, or interpreted, applications are native applications that uses an interpreter, that is bundled with the application on the mobile de-

[1] *TRIGGER.IO*, `http://trigger.io/`, Accessed: 2016-02-07
[2] *Ionic*, `http://ionicframework.com/`, Accessed: 2016-02-07

vice. The interpreter executes code during runtime to make calls to native APIs. This approach may use any scripting language which can be interpreted on a device, but most of the new frameworks use JavaScript as their primary language. Examples of native scripting frameworks are Appcelerator Titanium, NativeScript and React Native, which all use JavaScript. The three frameworks are available as open source and are welcoming the contributions of the public.

Advantages of the native scripting approach are mostly the same as cross-compiled applications, for example the use of native components as well as sharing code between platforms. Since the interpreter is used to call native APIs everything that is possible in a native application is possible through native scripting. However this will result in a loss of performance compared to calling the native environment directly [5].

Different interpreters and frameworks support different features, platforms and platform-specific features. The features that are supported is a constant work in progress. Titanium has been in development since 2008, while both NativeScript and React Native are younger frameworks that were announced during 2015.

Platform-specific features will need platform-specific code and different frameworks use different techniques to make this available. All three frameworks can use plugins that can be imported into a project. If there is no written plugin available the frameworks use different techniques to extend this functionality. NativeScript uses an approach where no native code at all should be necessary, but all the native APIs are callable from JavaScript. Meanwhile both Titanium[3] and React Native[4] requires writing a native module which is imported. Both these approaches need a programmer that is familiar with native APIs and know how they work, but NativeScript[5] only needs JavaScript code.

All of the native scripting frameworks has the common denominator that they use an interpreter in some way, but besides that they can use vastly different techniques to call the native APIs. The biggest difference between Titanium, NativeScript and React Native is that Titanium and NativeScript use an MVC architecture while React Native uses an architecture that is inspired by the JavaScript library ReactJS, as described in section 2.1.5. [5, 7, 3]

2.1.5 React Native

React Native is a native scripting framework used for creating cross-platform mobile applications that was first introduced during the React.js conference

[3] *Titanium SDK Documentation* `http://docs.appcelerator.com/platform/latest/#!/guide/Titanium_SDK` Accessed: 2016-02-09

[4] *React Native: Documentation* `http://facebook.github.io/react-native/` Accessed: 2016-02-09

[5] *How NativeScript works* `http://developer.telerik.com/featured/nativescript-works/` Accessed: 2016-02-09

2015 [2]. In early 2015 React Native only supported development of iOS applications however, the framework was expanded to include Android support in September, 2015 [8]. React Native promises cross-platform development in the sense that large parts of application code can be shared between platforms even though some platform-specific code is required. Furthermore, it is an open source framework which allows for the programming community to contribute to its development.

React Native is built upon principles and concepts of ReactJS[6] which is a Javascript framework that was open-sourced in 2013, but Facebook has used it internally since 2011 [2]. ReactJS is sometimes mentioned as only React, but in this report it will be called ReactJS to not confuse the reader to mistake it for React Native. Even though React Native was released only recently it has an established core through ReactJS. The core concept behind React Native is "learn once, write everywhere". This means that if a developer can create a web application through ReactJS he or she should also be able to create React Native applications for all platforms without prior native development experience. React Native and ReactJS are very similar in code structure. Facebook, who created both frameworks, explains that the difference between them is that ReactJS operates on the Document Object Model(DOM), in a web browser, while React Native operates on the mobile application view. This also means that code written for mobile applications largely can be reused and shared to ReactJS web projects. [2, 3]

Internal structure

One of the most important features of both ReactJS and React Native is how the frameworks operate on the application view hierarchy when changes occur. There are three different traditional approaches to handle these changes in web development. One way is to send a new HTML request to the server in order to re-render the entire page. The second is to use client-side HTML templating that will re-render partials of the page. The third, and most efficient, way is to make imperative HTML DOM changes. The two former approaches can cause delays as large parts of the DOM, or even the entire DOM, needs to be re-rendered which can be an expensive and slow operation. However, these approaches often allow good code strucutre with code that is easy to read and maintain. Manually changing the DOM-tree through imperative programming is a faster approach, but during development of larger applications this technique is often error prone and hard to maintain[7].

ReactJS[8] attempts to use the advantages of the aforementioned approaches and work around the disadvantages by using a virtual DOM. Reac-

[6] *React: Releases*, `https://github.com/facebook/react/releases?after=v0.9.0-rc1`, Accessed: 2016-02-10

[7] *Vue.js Overview*, `http://vuejs.org/guide/overview.html`, Accessed: 2016-02-10

[8] *React: A JavaScript Library For Building User Interfaces*, `https://facebook.github.io/react/`, Accessed: 2016-02-11

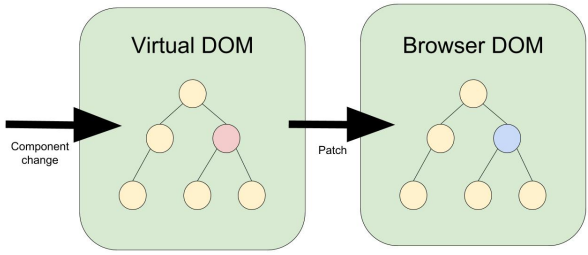

Figure 2.1: When a component changes the virtual DOM will create a patch of imperative DOM operations that is sent to the Browser DOM.

tJS uses its components to serve the same purpose as templates traditionally do, to structurally swap out blocks of code when an event occurs. An important difference between templates and ReactJS components is that instead of updating the browser DOM directly and replacing the old content, which invokes a re-rendering of the entirety of the affected area, the updating of components will trigger a re-calculation of the virtual DOM and result in a patch that is sent to the browser DOM. The virtual DOM is never rendered in the browser which saves a lot of time as re-rendering the DOM through a template usually is time consuming while calculating the virtual DOM is a relatively cheap operation. The virtual DOM that is updated with the change of a component is then compared to the browser DOM and the least amount of necessary changes to convert the browser DOM into a copy of the virtual DOM is calculated. These changes are then queued as a patch and added to the browser DOM asynchronously, through imperative DOM manipulation as can be seen in figure 2.1. A virtual DOM is effective, since the imperative DOM manipulations are fast. React Native uses the same approach as ReactJS when updating the application, however instead of a virtual DOM it operates on a virtual application hierarchy. Since the React Native calculations are flushed onto the main thread each render there is no longer a need for recompiling the entire application whenever a change is made. Instead, when the code is changed React Native will simply apply the necessary changes using the virtual application hierarchy.

To make ReactJS code easier to read and write Facebook has developed a JavaScript syntax extension, that looks similar to XML, called JSX. A compiler that supports JSX can then compile JSX code into regular JavaScript code, but it is not a requirement to use JSX just a possibility. React Native is shipped with a compiler called Babel[9] which supports JSX and Facebook highly recommends writing JSX code. Babel also supports ES2015 which makes it possible to use the newest JavaScript syntax and features without having to wait for interpreter support. An example of ReactJS code written

[9] *Babel the compiler for writing next generation JavaScript*, `http://babeljs.io/`, Accessed: 2016-02-11

Figure 2.2: Hello world written in JSX and regular JavaScript.

using JSX and JS is available in Figure 2.2.

React Native communicates with native APIs through a JavaScript bridge with JavaScriptCore[10] as its interpreter. The JavaScript bridge connects the JavaScript side and the native side of the application, as seen in figure 2.3. The JavaScript side runs on a separate asynchronous thread from the main thread and does not interfere with the native UI. When a call is made from the JavaScript side of the application the call is stored in a message queue if it can not be sent immediately. Before sending the information to the native side the JavaScript bridge will convert JavaScript data types to match native data types automatically. The native mobile main loop runs at 60 frames per second, which the JavaScript bridge matches in order to maximize performance. The method return types of the bridge can only be void, hence to pass information from the native side to the JavaScript side events or callbacks needs to be used. Event listeners can be implemented in a number of ways but a simple solution is to create an event emitter on the native side[11] to send the event and to create a listener for the component in JavaScript[12]. Callbacks are a special data type that is defined with any number of out parameters that the native side will have to set when the callback occurs[13]. These callbacks are stored on the JavaScript side.

React Native runs on two threads as well as with additional dispatch queues. The native UI runs in the main thread while the second thread is a JavaScript thread[14] that runs the JavaScript code of the React Native application. Every native component also uses a Grand Central Dispatch Queue[15] (iOS) or a dedicated MessageQueue[16] (Android) to handle threads

[10] *React Native: JavaScript Environment*, `https://facebook.github.io/react-native/docs/javascript-environment.html#content`, Accessed: 2016-02-10

[11] *React Native: Native Android Modules*, `http://facebook.github.io/react-native/docs/native-modules-android.html`, Accessed: 2016-02-11

[12] *React Native: Native iOS Modules*, `https://facebook.github.io/react-native/docs/native-modules-ios.html`, Accessed: 2016-02-11

[13] *React Native: Native Communication between native and React Native*, `https://facebook.github.io/react-native/docs/communication-ios.html`, Accessed: 2016-02-11

[14] *React Native: Performance*, `https://facebook.github.io/react-native/docs/performance.html`, Accessed: 2016-02-11

[15] *iOS Developer Library: Dispatch Queues*, `https://developer.apple.com/library/ios/documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationQueues/OperationQueues.html`, Accessed: 2016-02-11

[16] *Android Develop Reference: Handler*, `http://developer.android.com/reference/`

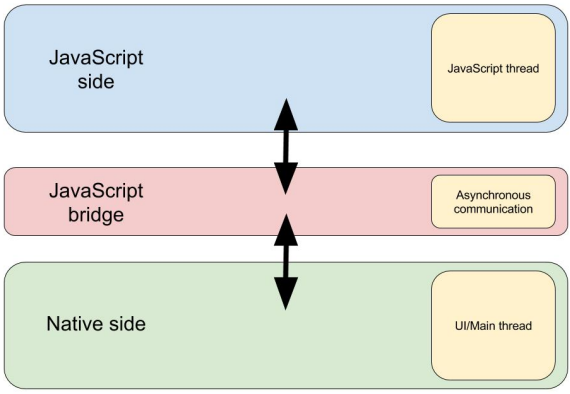

Figure 2.3: The JavaScript side runs with a JavaScript thread and communicates asynchronously through the JavaScript bridge with the native side which runs on the main thread.

and concurrency.

Application structure

A React Native application is represented as a tree of composite components where each composite component has a render function that returns a subtree of virtual application hierarchies. Each composite component stores the internal state and listens for state changes. When a state change occurs the component will get the change event and re-render itself. A composite component is a custom React Native class that wraps native components that has been implemented in React Native. This way the React Native application will actually consist of native components since React Native components are implemented in native languages and uses the native SDKs. A lot of the most common components are already implemented in React Native but it is also possible to create new, customized components through the RCTBrigeModule protocol and native programming with a React Native markup syntax. The component tree structure of React Native makes the application highly modularized. [1]

To layout the application hierarchy React Native uses an implementation of CSS3 Flexible box, or Flexbox[17], which is a layout mode for arranging components. The core concept of Flexbox is that flex items in a flex container will be able to automatically fill the container, or shrink in order to stay inside it, without affecting the layout of neighboring containers. Each

`android/os/Handler.html`, Accessed: 2016-02-11

[17] *Using CSS flexible boxes*, `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Using_CSS_flexible_boxes`, Accessed: 2016-02-11

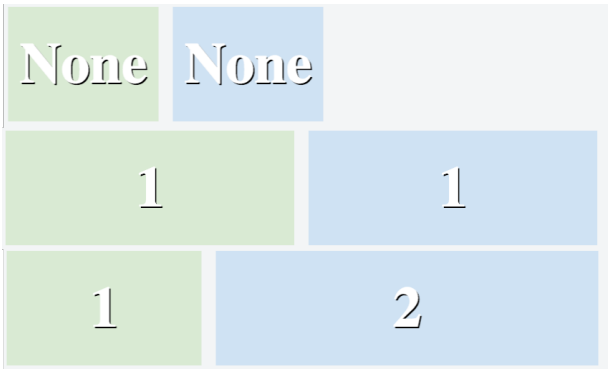

Figure 2.4: An illustration of how different flex values work.

flex item has a flex value which decides how much of free space that should be allocated to this item. As shown in Figure 2.4 an item will flex proportionally to other items in the same flex container. If several items has the same flex value they will share the free space equally, but if one has a bigger flex value than the other it will take more space in proportion to the flex values. React Native has implemented a subset of the Flexbox layout model to layout components as well as support for common web CSS styles. Because of Flexbox React Native can handle different resolutions and screen sizes, since content will fill up space or shrink in order to fit and this is a big problem in cross-platform development. [3]

2.2 Android development

Android is an open platform that includes an operating system, user-interface and built-in applications. Android was developed by the Open Handset Alliance[18], a collaboration group now consisting of 84 technology and mobile companies. The members of the Open Handset Alliance represent the entire mobile ecosystem and they all believe that everybody benefits from an open, free mobile platform. The industries represented in the alliance includes mobile operators, handset manufacturers, semiconductor companies, software companies and commercialization companies. Together they developed the first widely available Android version, Android 1.0 which was released in September 2008[19]. The original Android concept was created by a standalone software company *Android, Inc* in 2005 who later that year was bought by and started a collaboration with *Google Inc* to develop the early versions of the Android platform. *Google Inc* now leads the Android Open

[18] *Open Handset Alliance Overview*, `http://www.openhandsetalliance.com/oha_overview.html`, Accessed: 2016-02-15

[19] *Android Timeline*, `http://faqoid.com/advisor/android-versions.php#version-1.0`, Accessed: 2016-02-15

Source Project and the latest released version, 6.0, is called "Marshmallow". This version was released in December 2015.

To develop native Android applications the language Java is used and the development environment includes the Android SDK, as well as the Java JDK. There are several IDEs that can be used to develop Android applications, but Google recommends Android Studio. It is based on the IntelliJ IDEA software which is developed by JetBrains [20]. To debug the applications Google provides an emulator that can emulate any Android device, but applications can also be installed and run on a physical Android device. [21]

Google also provides a tool for analysis and application debugging called the Device Monitor. It includes several tools, including Dalvik Debug Monitor Server (DDMS) and Systrace. These can be used to measure CPU usage, memory usage and response time on Android devices and emulators. [22] [23]

2.2.1 OS structure

The Android platform consists of many layers and is based on the Linux kernel. The Android architecture is viewed as a set of layers that together form the Android platform stack as visualized in figure 2.5. Android applications rely on the application framework where an API to core components can be accessed. In most cases application developers will not have to go any deeper than this high-level framework in order to access the underlying Android functionality. The access is enabled by the Binder Inter-Process Communication (Binder IPC) mechanism that allows for the system calls to be made from the client to, for instance the window manager or activity manager. The window manager also uses an instance of the binder in order to communicate with the low-level surface compositor and will indirectly allow for the client to talk to low-level components in the Android stack.

Components, such as the mentioned window manager and activity manager, are considered to be in the system services layer, below the Android framework in the Android stack. The system services layer includes native libraries and Android runtime components. These middle components communicates with the Hardware Abstraction Layer (HAL) that provides an interface to lower-level drivers. In this way, lower level systems can be introduced that will be compatible with any higher level system as they will work independently when implemented with the HAL standard. A hardware device distributor is free to customize the communication between the hardware component drivers and the HAL whichever way suits best. However,

[20] *IntelliJ IDEA*, `https://www.jetbrains.com/idea/`, Accessed: 2016-02-15

[21] *Android Studio Overview*, `http://developer.android.com/tools/studio/index.html`, Accessed: 2016-02-15

[22] *Analyzing UI Performance with Systrace*, `https://developer.android.com/tools/debugging/systrace.html`, Accessed: 2016-02-15

[23] *Using DDMS*, `http://developer.android.com/tools/debugging/ddms.html`, Accessed: 2016-02-15

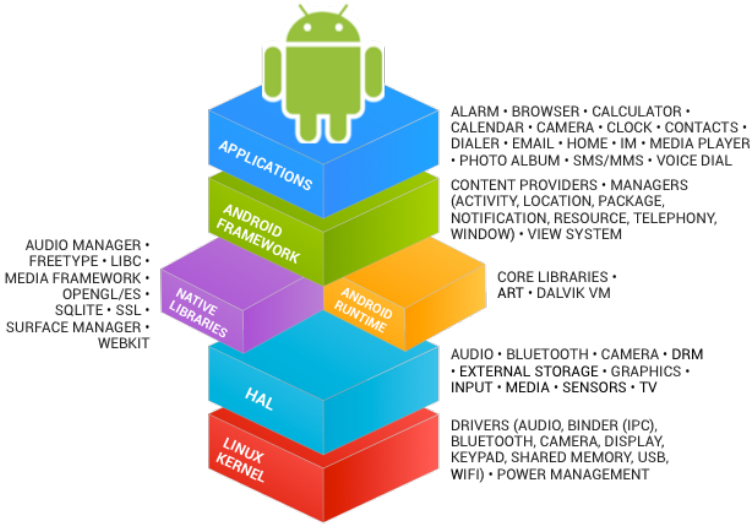

Figure 2.5: The Android stack. [9]

the communication between the HAL and the system service layer must follow a standard with implementation specifications described in a meta file. Each hardware component in a mobile device has its own instance of an HAL which most commonly extends a generic module structure. At the bottom of the Android stack is the Linux kernel which serves as the core of the Android platform. Android implements a version of the Linux kernel that has been extended with the Binder IPC driver and other drivers that are helpful in a mobile environment. [24] [25] [26]

2.2.2 Application structure

Android applications are built using different application components. There are four different kinds of components: Activities, Services, Content Providers and Broadcast Receivers. Each of these components have a different purpose and exists as its own entity. Together they form the application and define its behavior. Every application needs to declare its components in the application manifest file. This manifest file also defines what permissions the application needs, what OS version is needed to run the application, what hardware components are used by the application as well as if any Android libraries are used in the application.

[24] *The Android Source Code*, `https://source.android.com/source/index.html`, Accessed: 2016-02-15

[25] *Android Interfaces and Architecture*, `https://source.android.com/devices/index.html`, Accessed: 2016-02-15

[26] *Android Binder*, `http://elinux.org/Android_Binder`, Accessed: 2016-02-15

Application Components

- An Activity is a single screen with a user interface. Each Activity should function on its own and be able to be started from another application. Together all the Activities form the user experience of the application, since they represent all the UIs and transitions between them.

- Services are components that run in the background of the application without user interfaces. They run background tasks such as downloading a file or playing sound in the background. The Services are started and interacted with by other components, such as activities.

- Content Providers are components for data handling in an application. They are used to save and modify data concerned by applications. Each Content Provider can be setup to allow different applications to modify data handled by itself. For example, the provider that handles the Android address book can be accessed if your application manifest explicitly requests access to this provider. The Content Providers can also be used to handle data that is private to only one application.

- Broadcast Receivers handle system-wide broadcasts that can be accessed by all applications. The system broadcasts messages when, for example, the screen turns off or the battery is low. Applications that subscribe to this message can then modify their behavior in regards to this event. Usually these components do not perform much work, but rather starts a service when a specific event is received. However, they do have the possibility to create status bar notifications if the user needs to be notified about something application specific.

Since all the components are independently functional every application can start a component of another application. For example, if an application needs to use the camera, it can start the Android system camera activity. Since each application is run as its own process it does not have the required permissions to start this activity on its own. Instead it uses the Intent System to provide the system with the activity and the purpose of it. After that the system starts the needed component in another process and the started component does not belong to the process that requested the Intent. When the component has finished its work it will close and send the requested data back to the application that requested the Intent. One powerful feature of the Android system is that implicit Intents can be used. Implicit intents means that actions that can be performed by an activity are declared in the manifest file. If another application then notifies the system that it wants to perform this action the user will be able to choose from all the applications which offer an activity for this kind of action.

Since any component can be started from any application Android applications does not have a main function like many other programs. Instead

each component has its own lifecycle that handles different events. The activity lifecycle handles how activities should act when another activity is opened or closed. When another activity is started the current activity is pushed onto a stack of opened activities. The system will keep the activity state in memory until it is popped back from the stack. [27]

Activity lifecycle

An activity has three different states: resumed, paused and stopped. A resumed activity is running in the foreground of the application and is the current activity. A paused activity is running in the background and is partially visible on screen beneath the running activity. A stopped activity is not visible on screen, but is still alive in memory. Both stopped and paused activities can be shut down by the system in case of low memory situations. When an activity has been shut down and needs to be restarted it will not have any saved state, but will instead be created all over again.

Each activity implements 6 different callbacks that are called during different points of the activity lifecycle, as illustrated in figure 2.6. The first call is made to onCreate when the activity is created and the last call is made to onDestroy when the activity should exit, and release all used resources. When the activity receives the onStart callback the user can now see and interact with it and the activity should handle events. Similarly the onStop callback is received when the activity is no longer visible to the user. These callbacks can be called several times during an activity lifecycle when the user shows and hides the activity. When the onResume callback is received, the activity is in front of all other activities and has user input focus. Whenever the activity is hidden, for example if the screen is locked the onPause callback will be received. When the activity comes back into user focus onResume will be called. The cycle between these two states is frequently revisited during an activity lifecycle. [28]

2.3 iOS development

iOS is a operating system developed by Apple for exclusive use in Apple hardware products. It was unveiled in 2007 during the iPhone release and has been adapted for, among others, the iPad and the Apple TV. The OS has been updated several times since its release and the newest version, iOS 9.2, was released in December, 2015.

There are several requirements that needs to be fulfilled to be able to develop applications for iOS. The development environment includes a Mac computer with OS X 10.10 or later installed and the Apple IDE Xcode.

[27] *Android Application Fundamentals*, `http://developer.android.com/guide/components/fundamentals.html`, Accessed: 2016-02-15

[28] *Android Activity Lifecycle*, `http://developer.android.com/guide/components/activities.html#Lifecycle`, Accessed: 2016-02-15

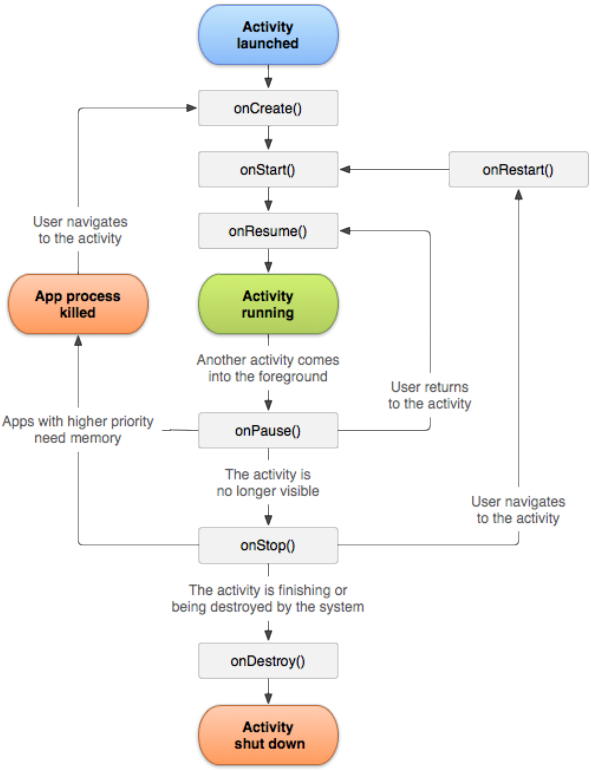

Figure 2.6: The activity lifecycle in an Android application [10].

Included in Xcode is the iOS SDK which is needed for iOS development[29]. Development languages are Objective-C or Apple's new language Swift, that was released in 2014. The latest Swift[30] update 2.1.1 was released in December 2015. A combination of both languages can also be used if needed as Swift is designed for interoperability with Objective-C.

Bundled with Xcode is also a simulator that can simulate an application on any iOS device, but the application can also be tested on a physical device. There are some restrictions in regards to what features that can be enabled in the simulator, for example push notifications are disabled.

Instruments is a performance analysis and testing tool that is bundled with Xcode. Using this an application can be tested in terms of CPU usage, memory usage as well as UI performance. The testing can be performed using either a physical device or the simulator. [31] [32] [33]

2.3.1 OS structure

The OS is structured into four different layers of frameworks[34]: Cocoa Touch, Media, Core Services and Core OS. The layers are built on top of each other and Apple recommends developers to use frameworks from the top layer if possible. These provide abstractions that reduces the amount of code needed as well as encapsulates potentially complex features. The layers are illustrated in figure 2.7.

The Cocoa Touch layer provides frameworks that define the appearance of the application as well as basic infrastructure and high-level system services. If possible a developer should try to only use frameworks in the Cocoa Touch layer.

If the application needs some sort of sound or graphics the developer should look into the Media layer, since it contains technologies for audio, video and graphics. These are frameworks that encapsulates complex tasks regarding media providing easier APIs to make applications look and sound the way they should.

[29] *Start Developing iOS Apps*, `https://developer.apple.com/library/ios/referencelibrary/GettingStarted/DevelopiOSAppsSwift/`, Accessed: 2016-02-09

[30] *Swift. A modern Programming language*, `https://developer.apple.com/swift/`, Accessed: 2016-02-09

[31] *Measure CPU Use*, `https://developer.apple.com/library/watchos/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/MeasuringCPUUse.html`, Accessed: 2016-02-15

[32] *Measure Graphics Performance*, `https://developer.apple.com/library/watchos/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/MeasuringGraphicsPerformance.html`, Accessed: 2016-02-15

[33] *Monitor Memory Usage*, `https://developer.apple.com/library/watchos/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/MonitoringMemoryUsage.html`, Accessed: 2016-02-15

[34] *About the iOS Technologies*, `https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html`, Accessed: 2016-02-09

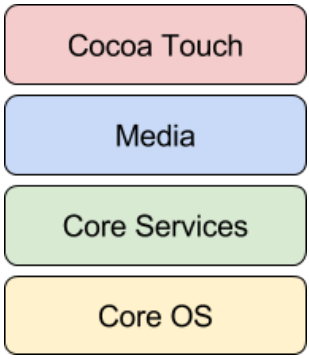

Figure 2.7: The four iOS framework layers in order.

The Core Services layer concerns the core system services needed in iOS applications. These services are for example: networking, location and social media services. One framework that resides in the Core Services layer is the JavaScript Core framework, which allows iOS to evaluate JavaScript code.

The Core OS layer contains the most low-level features that other frameworks encapsulate and use. This is the lowest level layer and most developers will not need to use these frameworks, but can rely on support from higher level layers.

2.3.2 Application structure

For iOS applications Apple recommends the Model-View-Controller (MVC) architecture[35], which separates data and business logic from presentation. This structure simplifies the handling of different resolutions and screen sizes, since the view component can be altered without affecting data and business logic. The structure is illustrated in figure 2.8

The model handles data and notifies the controller when changes in the data occur. In iOS this is done using data objects, which can be for example a database, but there are also an abstraction called document objects that can be used. Document objects should be used when different data objects are grouped together and the document object acts as a mediator in between data objects and the controller.

Views and UI objects are the visual representations of the content in an application. The view handles presentation of data and also notifies the controller when user actions occur. Each application has at least one UIWindow object which coordinates the views on a single screen. If an application uses several screens, for example an external display, several UIWindow objects are needed. A view is always a rectangular area which draws content and

[35] *The App Life Cycle*, `https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/TheAppLifeCycle/TheAppLifeCycle.html`, Accessed: 2016-02-10

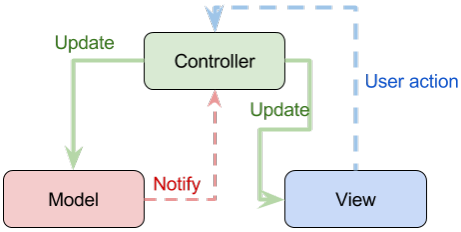

Figure 2.8: The MVC architecture.

responds to events inside this area. The UIKit framework provides standard views, but it is also possible to create custom views. Standard user interface components, like buttons or switches, are available as control objects. A UIWindow is constant throughout an application while the views are reusable components.

The controller acts as a mediator in between model and view. It consists of the UIApplication object, an app delegate object and view controller objects. The UIApplication object, that should be used as it is, handles the event loop and acts as a reporter to the app delegate object. The app delegate object is a custom object that handles state transitions, through communication with the UIWindow, and app initialization. It is the only object that is guaranteed to be present in every application. View controller objects controls a single view and all of its subviews as well as communication with the data model. It receives event data from the app delegate and updates views and models accordingly. There are several standard implementations of view controllers for standard views, for example the tab bar interface, but the view controllers can also be custom made.

2.3.3 Application state transitions

An iOS application state follows specified transitions as seen in figure 2.9. The possible states are: not running, inactive, active, background and suspended. When an application is not running it will not execute any code as it is not started or was previously terminated by the system. When a user launches the application it will enter the foreground and the inactive state.

The application is usually only in the inactive state briefly before it transitions to another state. If the application continues to run in the foreground it transitions to the active state. In the active state the application will execute code and handle events. Before an application transitions to the background it will always transition to the inactive state. When an application is in the background it can still execute code. This state can be used when an application is processing a request that it needs to finish before being suspended. An application that, for instance, needs to track the location of the user can be in background mode for an extended period of time.

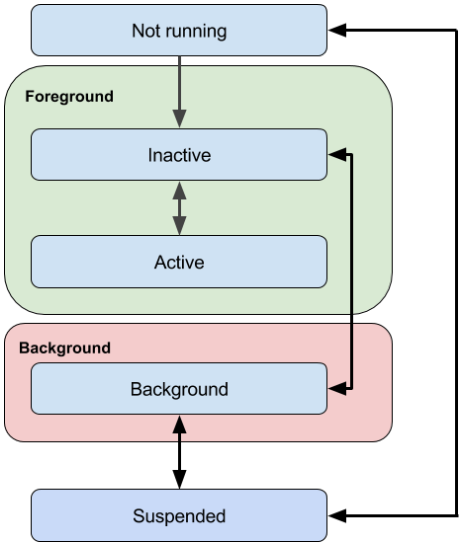

Figure 2.9: The iOS application state transitions.

When the expiration time for background mode is reached the system will move the application into suspended mode. In suspended mode no code can be executed, but the application remains in memory. In case the system needs to free memory it can terminate any suspended application to achieve this. Because of this every application must be ready for termination and save user data as soon as it is removed from the foreground.

All state transitions are tracked and handled by the app delegate object. If an application needs more time to finish a request, it is the app delegate object that notifies the system about this and requests a longer expiration time for the background state.

2.4 Evaluation techniques

2.4.1 Performance measurement studies

There are a lot of different ways to measure performance and several parameters which can be used as benchmarks for evaluating if an application is effective in comparison to others. This section will describe measurements of power usage, CPU usage, memory usage and responsiveness.

Power usage

The power usage of an application is defined as the sum of all battery consumption caused by the application. The factors that have the greatest impact on power usage in mobile applications have been shown to be the

use of display, wireless communications as well as the use of external sensors, like camera or GPS. Display usage and wireless communications are not affected by the development strategy, while use of external sensors could be impacted. However, the impact external sensors have compared to total power usage is so small that it can be ignored [4]. Other factors that can affect power usage, like CPU usage, can be measured more accurately than measuring actual power usage. [4]

CPU usage

CPU usage is defined as the percentage of total CPU capacity that is used by an application in a specified time interval [4]. CPU usage can be measured during different phases of application runtime, for example during startup. Usage of cross-platform tools will introduce additional overhead to the application and this could raise CPU usage. The collection of data can be event-driven, however such a data set can be hard to draw conclusions from depending on the frequency of the chosen event triggers. To achieve a result that better reflects the reality the data should be collected with a high sampling frequency [7]. The CPU usage is very relevant as a metric, as high CPU usage could impact other applications that is running on the device.

Memory usage

The amount of memory that needs to be allocated for the application is defined as its memory usage. This is often measured in percentage of total memory, as it has a higher impact on devices with smaller total memory. Memory usage is often different depending on the state of the application, if it is in the foreground or the background for example. Therefore several memory usage measurements should be taken to get a fair result. When measuring memory usage on the Android OS it is important to be aware that some of the memory usage of the application may be in memory pages that are shared with other applications. Therefore, when measuring memory usage it is important to realize the difference between Unique Set Size (USS) memory and Proportional Set Size (PSS) memory. USS denotes the amount of memory that is uniquely dedicated to the application. PSS shows the amount of unique memory as well as the portion of shared memory that is used by this application. [7] [36]

Responsiveness

The responsiveness of an application can be measured through response times from performing an action or through checking the UI thread frame drop rate. The response time of an action is defined as the elapsed time

[36] *Investigating Your RAM Usage*, `http://developer.android.com/tools/debugging/debugging-memory.html`, Accessed: 2016-02-18

from when an action is performed to when an expected result is achieved. The expected result of an action can, for instance, be for new information to be visible in the UI or for a certain event to occur.

Native applications will always try to run in 60 Frames Per Second (FPS) to give the user an experience without screen stutter. One of the main goals of React Native is to deliver a native experience, in other words, to deliver 60 FPS[37]. If any frames are dropped this should be regarded as a bug that needs to be fixed.

A user experiences an action to be instant if the response time is less than 0.1 seconds. Similarly, the response time needs to be below 1 second for the user to stay focused on the application and not start thinking about something else. If a task takes 10 seconds or more a user will want to perform other tasks during the completion. [11]

2.4.2 Look and feel user study

A user study can be conducted to evaluate the look and feel of a mobile application. There are several different questionnaires that focus on usability testing of an application. Two of these are User Experience Questionnaire (UEQ) and System Usability Scale (SUS). [12].

UEQ

The UEQ is a questionnaire that was created in Germany in 2005. It consists of six scales that describe different aspects of the user experience.

- Attractiveness - The overall impression of the product, do users like the product?
- Perspicuity - How easy is it to learn how to use the product?
- Efficiency - Can users solve given tasks efficiently using the product?
- Dependability - Do users feel like in control of the product?
- Stimulation - Do users feel motivated and happy using the product?
- Novelty - Is the product innovative and exciting?

Each item in the UEQ is a question regarding how the user felt about the product. It is represented by two terms, which are opposites, and seven circles. Each circle represents a value between -3 and 3 and the middle circle is a neutral answer. The order of the terms are randomized so that half of the items for each scale has a negative term to the left and half of the items has a positive term to the left. Users shall be instructed to respond according to their first thought, and not think about answers for too long. If a user can

[37] *React Native: Performance*, `https://facebook.github.io/react-native/docs/performance.html`, Accessed: 2016-02-11

not answer a question, the middle point of the scale should be checked. The evaluation should be conducted directly after usage of the product to catch the user's immediate impressions of the it. Discussion about the product should be saved for after the evaluation.

If changes are to be made to the UEQ there are some restrictions to keep in mind. If an item is to be removed from the questionnaire, then all of the items included in the corresponding scale of the removed item also needs to be removed, essentially removing the entire scale from the test. This means that the questionnaire can be reduced by a certain scale, but not by parts of it. If only parts of a scale is removed, the results can no longer be compared to previous results using the UEQ. Hence, if a certain scale is not interesting it can be removed and the results will still be valid.

The UEQ requires different amounts of data to give reliable results. Typically this has been shown to be around 20-30 participants, but it depends on the results. If the standard deviations of item answers are high, then more data is required to achieve statistically proven results. The results can then be compared between applications through comparisons of the different scale mean values. [13, 14]

The results of the UEQ can be run through the UEQ data analysis tool, which generates mean and confidence values as well as Cronbach coefficients for each scale. The confidence value is based on a selected α-value, the standard deviation for the scale as well as the number of respondents. The confidence value is used to calculate the confidence interval, which is the interval between $mean - confidence$ and $mean + confidence$. This interval is the range where $x\%$ of all answers are expected to be. The x value depends on the chosen α-value, for example $x = 95$ if $\alpha = 0.05$ and $x = 80$ if $\alpha = 0.2$ etc. [13]

The Cronbach coefficient is used to evaluate the correlation of related answers. For questions such as: annoying or enjoyable and good or bad, the answers should be similar. If a user rates the application as both annoying and good it is probable that the user did not interpret the question correctly. This would result in a low Cronbach coefficient for this scale which means that the results for this scale should be interpreted with care. If the respondents to a UEQ answers correlated questions similarly this will result in a high Cronbach coefficient for this scale. A good rule of thumb is that if the Cronbach coefficient is less than 0.6 the results should be interpreted carefully. [13]

The UEQ also comes with a comparison tool which can be used to compare two different products in regards to user experience. This tool produces graphs and can also be used to conduct a two sample t-test assuming unequal variances. This is used to control if the results have a significant difference, i.e a difference that is not based on randomness. The t-test also uses a selected α-value which controls how statistically certain the result is. For example if $\alpha = 0.05$ and a significant difference is found this means that with 95% probability the result is not due to randomness. [13]

SUS

The SUS is a questionnaire that was created by John Brooke in 1986. It was designed to be a "quick and dirty" usability scale that, with a low cost, could compare usability between systems. The scale focuses on the following three aspects of usability:

- Effectiveness - How well the user can complete given tasks using the system.
- Efficiency - How easy or hard it is to perform the tasks given.
- Satisfaction - The users subjective view of the system.

These aspects are reduced into a 10-item questionnaire. Each item is a statement and the user shall define how much they agree with each statement. The scale of the answers are 1 to 5, where 1 means "Strongly disagree" and 5 means "Strongly agree". The items were chosen from a pool of 50 potential items that were evaluated on two different systems by 20 users. The 10 items that produced the widest range of results were then chosen to make the SUS. The items are alternated positive and negative items, to make sure that the user must read and think about each statement. All items needs to be answered and if a respondent can not answer an item they should choose the value 3.

The questionnaire should be taken after the user has interacted with the system for a while and completed some tasks, but before discussing the application with anyone else. The respondent should not think about the answer for each question for too long, but evaluate it in regards to their first instinct.

The result of SUS is a score that is calculated using a specific formula. It is ranged in between 0 and 100, but should not be looked at as percentages. Total scores can then be compared between applications to draw conclusions. [15]

2.5 Related work

Other user experience comparisons of cross-platform mobile development has been performed by the following papers. [12] uses a modified version of UEQ and SUS comparing native versions to the cross-platform framework Titanium. [16] compares how well PhoneGap, Titanium and Intel XDK integrates with analytics as well as an undefined UX expert evaluation. [17] performs a comparison of the framework MoSync with native through their own definition of how satisfied users are with the different applications. A longitudinal study of PhoneGap versus native was performed by [18] where users tried different versions of an application and decided which they preferred.

Different kinds of cross-platform performance-measurements have been made by [4, 7, 19]. [4] focuses on CPU usage, memory usage, disk space and response times when comparing native versions with Xamarin and PhoneGap. [7] does a comparison between the frameworks Titanium and PhoneGap in regards to CPU usage, memory usage and Power consumption. [19] does a power consumption comparison between Titanium, PhoneGap and native when different sensors and hardware features are used.

Comparisons between different cross-platform approaches and frameworks have been made by [5, 6, 20]. [5] focused on what approach should be used depending on what kind of application that was developed while [6] did a comparison between Titanium, PhoneGap, mobile web applications and native applications. Game development for different mobile platforms were studied by [20] who did a comparison between the frameworks XMLVM, PhoneGap, PhoneXML, DragonRad and RhoMobile.

Chapter 3

Method

This chapter contains descriptions about how the results were achieved, which tools that were used and why it was performed this way. The development section presents the application concepts and development processes that were used. The performance evaluation section presents how the different performance measurements were made for different platforms. The platform code sharing section explains how the classification of shared code was made and lastly the look and feel user study explains how the user study was conducted and how the results were analyzed.

3.1 Development

This section presents a detailed description of the layout and functionality of the application as well as the task backlog that was used during the development on all platforms. Two versions of the application were developed using the native languages for Android and iOS, Java and Swift respectively, and one version was made using JavaScript and React Native. Furthermore, the React Native version was built with two OS-specific UI-configurations.

3.1.1 Application concept

The concept of the application was developed in collaboration with Attentec. The goal was to create an application that could stress the client in various ways and reach states where differences in development technologies may have a noticeable impact on performance or user experience. It was also important that the concept included enough features to be able to conduct a meaningful user study.

The application is a home automation application in conjunction with smart home devices, for example lamps and radiators. It consists of a client that communicates with a back end through a RESTful API. All the devices are merely virtual objects that are saved in a database.The user is able to

receive data and modify the settings for each device in the application. The user is also able to receive summarized information for a device, a room or the entire house. This data is presented using graphs.

UI concept

UI concept designs, that are available in appendix A, were created using the previously mentioned application concept. Two different OS-specific concepts were created since React Native is able to use real native components and design guidelines. The Android UI was designed to use the standard ViewPager pattern as well as the hardware back button. The iOS version uses the Tab Bar pattern for navigating between different views as well as a button, to navigate backwards, in the top left corner. Some small design differences, like the arrows on list items in iOS, were also adapted into the UI concepts.

Both the Android and the iOS version of the application will use four standard screens: the home screen, the room screen, the device screen and the stats screen. The concepts were designed without taking any features of React Native into consideration. This was done to make sure that the application followed native standards and guidelines, since that is one of the goals of React Native. The applications were developed to resemble the UI concepts as much as possible.

Feature backlog

A backlog with development tasks was written to make sure that all parts of the UI concept would be implemented. The backlog consists of a high level description of what features a user can expect to be available in the application. There are also tasks that describe what the developer expects the application to be able to do, such as communicate with a back end. The backlog can be seen in appendix C.

3.1.2 Development process

The tasks in the backlog were ordered according to priority and each task was visualized with a note on a board. The backlog tasks were given a priority to make sure that the most essential tasks for the thesis work were performed first. In this way, if there was not enough time to complete all of the tasks in the backlog, the resulting application would contain the most important features it needed to be evaluated and for the work to continue. The visual process management system Kanban was used to organize the work flow. Kanban was used because of the low overhead it brings to a project compared to other process management systems, for example Scrum. This was key as the time frame for the development of each application was short. [21]

3.2 Performance evaluation

A performance evaluation was conducted to compare the performance differences between native and React Native, as described in research question 1. The performance evaluation was divided into five different performance factors: CPU usage, memory usage, frames per second, response time and application size. These factors were chosen to give an extensive result. Some of the measurements have been used in related work and have shown to be problems for cross-platform frameworks. Each of these were evaluated using OS-specific application tools on Android and iOS. For Android the device that was used was an LG Nexus 5X with OS version Android Lollipop 6.0.1. The iOS device was an iPhone 5 running iOS 9.3.1.

Three different performance test scenarios, described in section 3.2.1 were created to measure CPU usage, memory usage and frames per second. The scenarios were selected to create a stress test for the device and to cover most of the functionality of the application. The response times of the applications were measured using a set of user interactions, described in section 3.2.5.

3.2.1 Performance scenarios

This section describes the selected performance scenarios as well as how automation was performed.

1. Expand both graphs on the home statistics screen, sequentially. Wait for 5 seconds.

2. Expand both graphs on a radiator statistics screen, sequentially. Wait for 5 seconds.

3. Select a device and flip the switch. Press the back button and flip the switch for the device again.

The performance tests were run using automated test scenarios, to make sure that all scenarios were performed in exactly the same way each time they were run. For Android devices, these were run using the AndroidViewClient tool [1], which is a tool for controlling an Android application outside of Android code. The tool is controlled through scripts written in Python. For iOS devices, the tests were run using the Automation tool [2], which is a part of the Instruments package that is bundled with Xcode. The Automation tool uses scripts written in Javascript to perform automated interactions. The source code of the scenarios, for both Android and iOS, is available in Appendix D.

[1] *AndroidViewClient*, `https://github.com/dtmilano/AndroidViewClient`, Accessed: 2016-04-13

[2] *Automate UI Testing in iOS*, `https://developer.apple.com/library/ios/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/UIAutomation.html`, Accessed: 2016-02-18

To make sure that the recorded data was reliable, all the scenarios were recorded five times each and average measurements for CPU usage, memory usage, frames per second and response time were then calculated. All non-system applications were terminated before the tests to make sure that they did not affect the results.

3.2.2 CPU usage

CPU usage was measured using the adb shell command top on Android and the Activity Monitor on iOS. Both are standard tools that are bundled with the standard IDEs, Xcode for iOS and Android Studio for Android.

The top command shows what percentage of the total CPU each running Android application is using and readings were taken once every second. This sampling time was chosen because of limitations in the top command. If a smaller sample time was chosen the result would be unpredictable and could contain both fewer or more sample points than expected.

Activity Monitor shows what percentage of the CPU that is used for each application on a iOS device. All the data had to be collected manually and the scenarios had lengths of 20-30 seconds. Because of this it was not feasible to use a sample time shorter than once every second, since each test was run 5 times each. The values were saved and analyzed in regards to mean, median, maximum and minimum values. [3] [4]

3.2.3 Memory usage

The memory consumption data was collected with the adb shell command dumpsys meminfo for Android and with the previously mentioned Activity Monitor for iOS. Due to similar limitations to those mentioned in section 3.2.2 the sampling time for both platforms were set to once every second.

The data was analyzed in regards to mean, median, maximum and minimum values. The data was also compared to the total amount of RAM of the mobile device, to conclude the percentage usage. [5] [6]

[3] *Measure CPU Use*, `https://developer.apple.com/library/watchos/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/MeasuringCPUUse.html`, Accessed: 2016-02-15

[4] *ADB Shell top*, `http://adbshell.com/commands/adb-shell-top`, Accessed: 2016-02-18

[5] *Memory Profilers*, `http://developer.android.com/tools/performance/comparison.html`, Accessed: 2016-02-18

[6] *Profiling Your App's Memory Usage*, `https://developer.apple.com/library/watchos/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/CommonMemoryProblems.html#//apple_ref/doc/uid/TP40004652-CH91-SW1`, Accessed: 2016-02-15

User interaction	Action	Result
1	Start the application	Home screen visible
2	Select a room	Room screen visible
3	Select a device	Device screen visible
4	Select the statistics tab	Statistics tab visible
5	Go back from device screen	Room screen visible

Table 3.1: User interactions for measuring response times.

3.2.4 Frames per second

During the scenarios the applications are supposed to keep the UI updated at 60 FPS. The actual performance of the applications were monitored and compared to each other and to this benchmark. To measure the FPS of the android application the tool Systrace was used. It records application usage for a duration and returns a trace of application events that can be analyzed [7]. This trace clearly marks every time a new frame is rendered and also marks a frame that took too much time to be rendered, also known as a frame drop. The resulting trace was analyzed and the average number of frames dropped and the frame drop percentage for each scenario was calculated.

For iOS no reliable tool was available and therefore no data was collected in regards to FPS performance.

3.2.5 Response time

The response time was measured as the time that elapses from when a user interaction is registered to when the expected result is visible. Some actions, like application startup, were used because they have previously shown to be of significance [4, 7]. The user interactions that were recorded are shown in table 3.1.

To measure the response time for Android, a trace of the application was recorded using the previously mentioned Systrace tool. This tool creates a trace of all the actions and events that the application emits and from this the trace response times could be concluded.

No suitable equivalent to the Android Systrace tool could be found for iOS. Instead, video recordings of the user interactions were used to obtain the response times. The video recordings were examined in Quicktime Player and the start and end of a user interaction was determined by looking at a recording, frame by frame. The time between the start frame and the end frame was then marked as the response time of the interaction.

[7] *React Native Android UI Performance*, `https://facebook.github.io/react-native/docs/android-ui-performance.html#content`, Accessed: 2016-05-02

3.2.6 Application size

Both the installer package size and the hard drive space needed for the installed application was measured to draw conclusions about application size. The Android application package file (APK) and iOS app store package (IPA) are the formats for the respective platforms. The APK and IPA files were created through the Android build system and Xcode respectively [8] [9]. To check how much hard drive space that was needed for an installed application, the built-in OS features were used. Both on Android and on iOS the installed application size can be found under the settings menu.

3.3 Platform code sharing

To answer how much code could be shared between platforms using React Native, as requested in research question 2, an evaluation of the React Native codebase was conducted. Each file was inspected manually to measure how much of it could be used for both platforms. The code was classified as either Android-specific, iOS-specific or shared. The percentage of shared code was then calculated. The difference in size between the Android- and iOS-specific code was also measured so that if one OS needed a larger amount of OS-specific code than the other, this would be taken into account.

This approach was chosen after research had been made to look for automated tools, like the one in [10] . However no such tool could be found for React Native.

3.4 Look and feel user study

To investigate the native look and feel, as research question 3 requests, the user interface was evaluated. The goal of the user interface evaluation was to answer if the React Native versions of the application have interfaces that can be considered equal to the native counterparts. To measure this, a user study was conducted on 32 people who were asked to perform a set of tasks. We chose to use the UEQ evaluation instead of the SUS evaluation, since it was easier to use when comparing two versions of a product. It was also chosen because it gives a clearer result which handles different aspects of usability.

Before a user performed the tasks he or she was asked if he or she was more comfortable using an Android or an iOS device. The user was then presented with an application with the OS that he or she felt most comfortable with. The user was not made aware if the tested application was a

[8] *Building and Running Overview*, `http://developer.android.com/intl/pt-br/tools/building/index.html`, Accessed: 2016-02-19

[9] *To create an Archived build that you can both test and submit*, `https://developer.apple.com/library/ios/qa/qa1764/_index.html`, Accessed: 2016-02-19

[10] *How mobile is your .NET?*, `https://scan.xamarin.com/`, Accessed: 2016-02-18

React Native application or a native application. Every other time the user was presented with the React Native application and every other time with the native application.

The user performed one task at a time in accordance with the list of user tasks, described in section 3.4.1. After completing all the tasks the user was immediately asked to fill in a questionnaire. The questionnaire was filled in by 14 iOS users and 18 Android users, where half used the native version and half used the React Native version. The selected questionnaire was the previously mentioned UEQ questionnaire, shown in Appendix E. This questionnaire was selected since it gives reliable results for a relatively small amount of users [13]. It is also a recommended approach for making a comparison of two versions of the same product.

In the tests, the user was required to set a score in the range 1 to 7 for each question. A question with a score of a high value can imply either a good or bad rating in regards to the scale that the question pertains to. To relate the questions to each other and to compile them into a combined score, the scale was transformed to the range -3 to 3.

When all the user tests had been completed, the results were run through the UEQ data analysis tool, which generated mean values, confidence values and Cronbach coefficients for each scale in the questionnaire. The UEQ comparison analysis tool was also used to generate graphs and conduct a t-test to check if there were significant differences between the results. The selected α-value to generate all the data was 0.05. The generated results were then compared between the two versions for each operating system.

3.4.1 User tasks

This section presents the tasks that users were asked to perform before filling in the UEQ.

1. Turn on the roof lamp in the bedroom.
2. Check the average temperature of the house.
3. Set the effect to 4 on the radiator in the living room.
4. Check the statistics of the living room.
5. Dim the light of a lamp to 78%.
6. Set the power consumption of the oven lamp to 6 W.

Chapter 4

Results

This chapter contains the results that were obtained from the evaluations described in the method chapter. It is divided into four sections: development, performance evaluation, platform code sharing and look and feel user study.

4.1 Development

The results of the development were four versions of the application, two running on iOS and two on Android. For each OS, there is one application written using native languages and one written using React Native. The resulting application screens are found in appendix B. These were supposed to resemble the UI concept images in appendix A.

The development results show applications that are very similar to the UI concept, for both platforms. Both React Native and native versions look very similar, even when they are compared side by side. Everey platform-specific feature from the UI concept is also present in each of the resulting applications. This was achieved with relative ease, and no feature was removed.

The hardest part to accomplish in development was the graphs on the statistics tab. These were constructed using a set of plugins that built on the MPAndroidChart-library [1]. This library was selected because it is available for both native Android and iOS as well as for React Native. Since the library was originally built for Android it was easier to customize the graphs for that platform. The React Native plugins were the least developed, but both of them are open source and built with Java or Objective C. This allowed for modifications to be made in the source code to further customize the graphs.

[1] *MPAndroidChart Github*, `https://github.com/PhilJay/MPAndroidChart`, Accessed: 2016-05-18

React Native did not contain any Android slider component and because of this a plugin that was created by the React Native community was used. This plugin was not using the animation features correctly and needed to be customized through modifications of the source code.

4.2 Performance evaluation

The following results were achieved from recording the automated test scenarios and the described user interactions. For each of the metrics, the two native applications were compared to their React Native counterpart.

4.2.1 CPU usage

The graphs in this section shows the percentage CPU load of the React Native application and the native applications for each scenario described in section 3.2.

The automated test runs for React Native and native Android or iOS starts off in synchronization, however after a varying amount of time the native version falls behind. Fortunately, it is easy to predict what CPU data that is the consequence of what input event thanks to the distinct spikes in data. This enables a fair comparison of React Native and native data.

Figures 4.1, 4.2 and 4.3, which represent CPU data from an Android device, all indicate a higher CPU usage on React Native than on native Android. The total average difference in CPU usage for Android is small, about 1-3%, but there are sample points where the difference is larger. These sample points, where the CPU usage spikes occur, are caused by user input events.

The iOS CPU usage in figures 4.4, 4.5 and 4.6 indicate that a React Native application on iOS also is more CPU intensive than a native iOS application. Once again, this is most noticeable at CPU spikes where the difference is about 10% in many cases. As for Android the average difference in CPU load between React Native and native iOS is small, about 1-3%.

React Native uses a lot of CPU at application start-up, which is indicated by the first spike in each figure. In the three scenarios, the CPU usage of React Native reaches values of 13-15% for Android and 82-85% for iOS. The mean value of the CPU usage in the three scenarios is in the range 4-5% for Android and 19-20% for iOS, which makes the React Native spikes very large. In comparison, the initial native spikes only reaches 5-7% for Android and 68-82% for iOS and the mean values are 3-4% and 16-18% respectively.

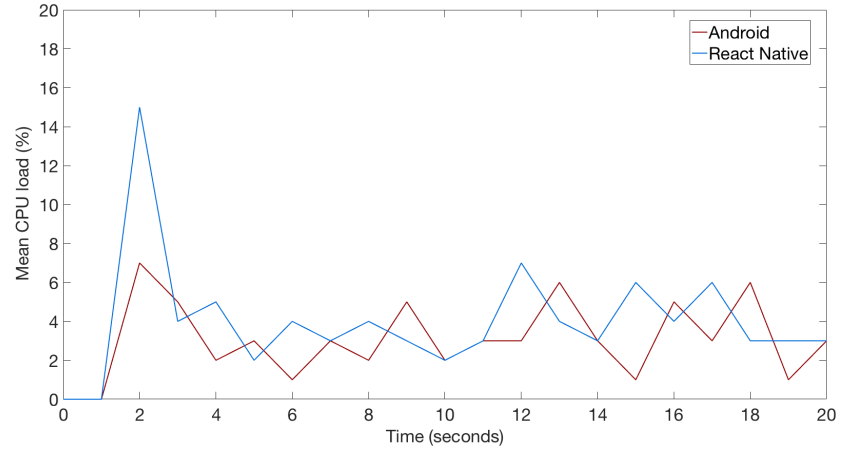

Figure 4.1: Mean % CPU load during scenario 1 for the React Native and Android application.

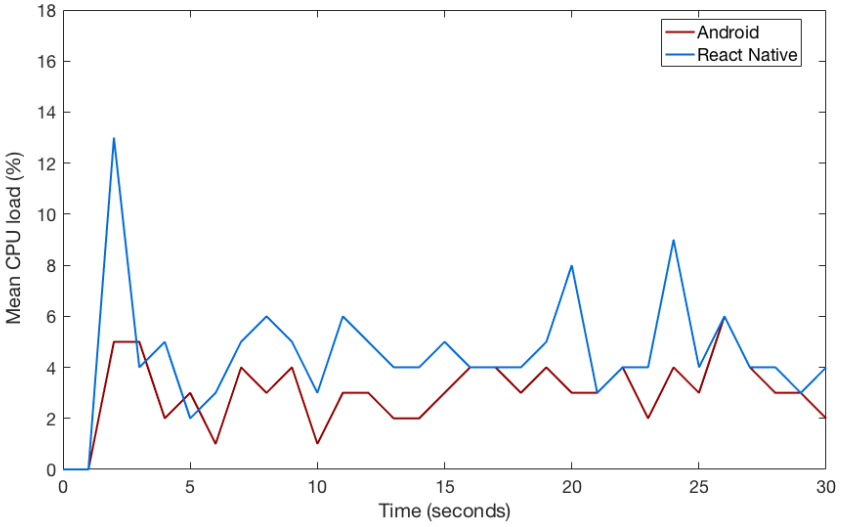

Figure 4.2: Mean % CPU load during scenario 2 for the React Native and Android application.

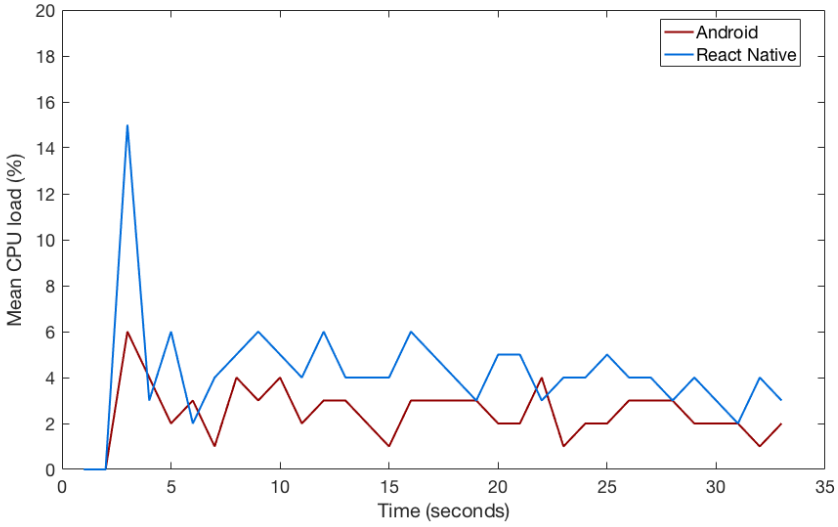

Figure 4.3: Mean % CPU load during scenario 3 for the React Native and Android application.

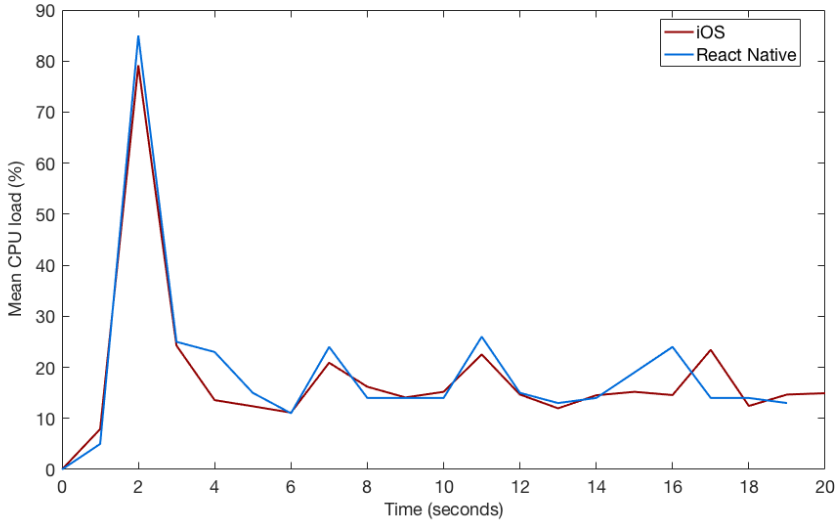

Figure 4.4: Mean % CPU load during scenario 1 for the React Native and iOS application.

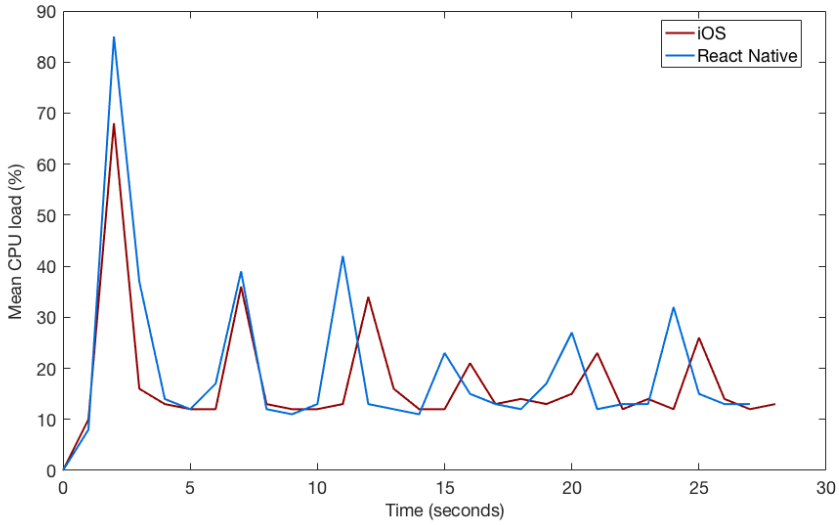

Figure 4.5: Mean % CPU load during scenario 2 for the React Native and iOS application.

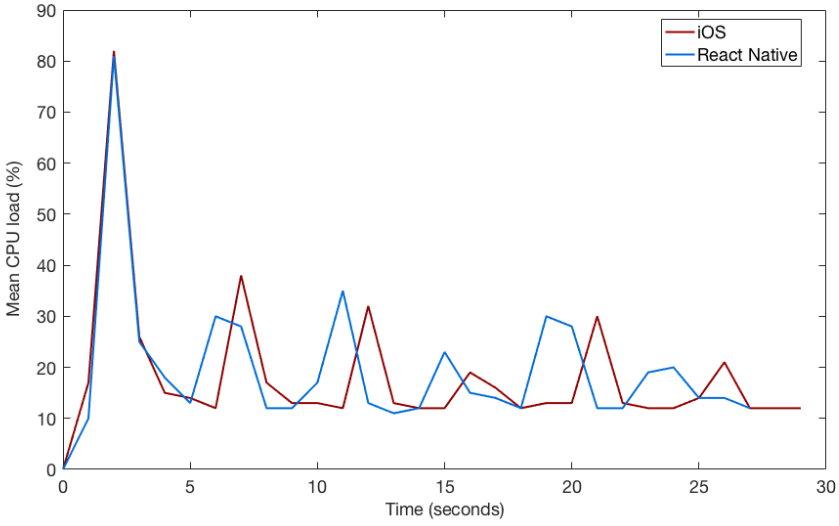

Figure 4.6: Mean % CPU load during scenario 3 for the React Native and iOS application.

4.2.2 Memory usage

Figures 4.7-4.12 shows the difference in percentage memory usage between the React Native application and the native applications for each scenario, described in section 3.2.

The figures 4.7 and 4.8 show results from similar scenarios, 1 and 2. These figures also show similar results where each rendering of a graph will result in a spike of memory usage. The React Native application spikes are higher than the respective native spikes.

Scenario 3, seen in figure 4.9, shows that React Native uses more memory at the start of the scenario, but also that the difference becomes smaller the more the native application is used.

The iOS results shown in figures 4.10, 4.11 and 4.12 are close to identical and the differences are small enough to be disregarded as errors in measurement or fluctuations between test cases.

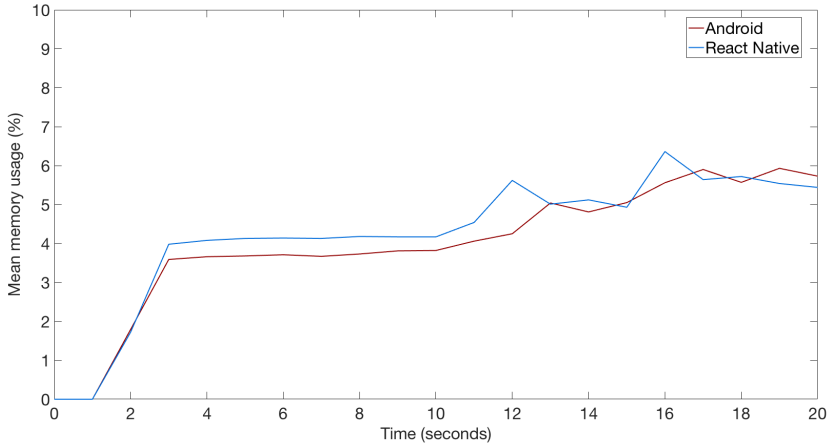

Figure 4.7: Mean % memory usage during scenario 1 for the React Native and Android application.

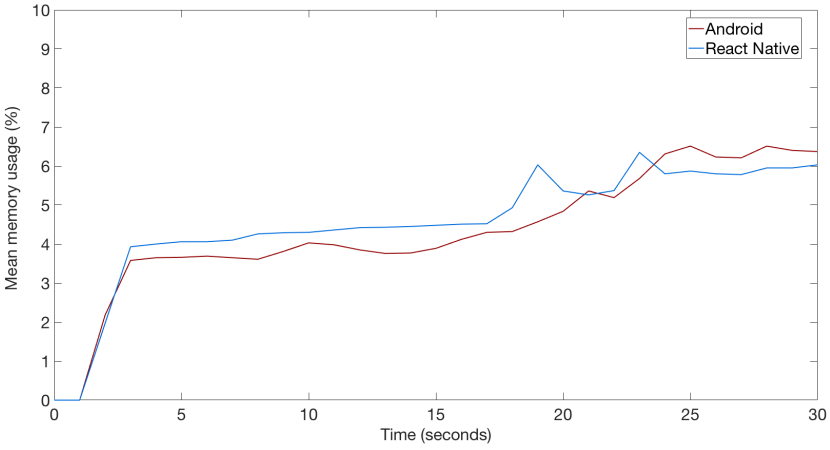

Figure 4.8: Mean % memory usage during scenario 2 for the React Native and Android application.

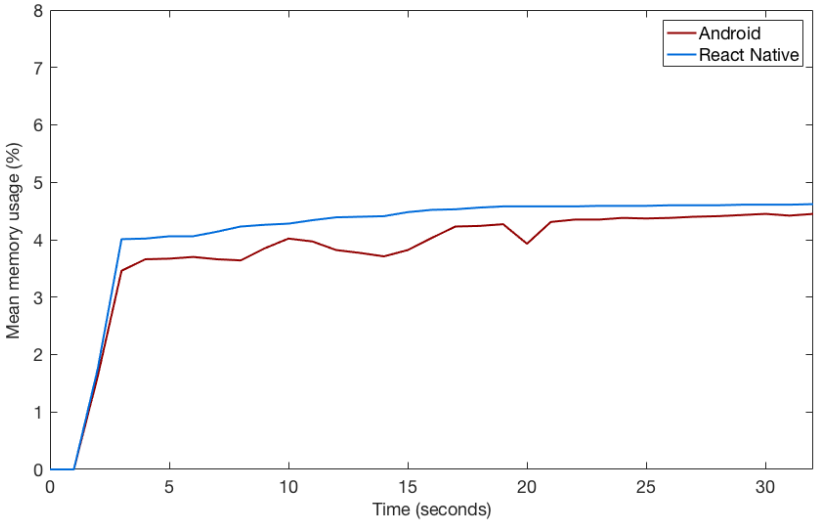

Figure 4.9: Mean % memory usage during scenario 3 for the React Native and Android application.

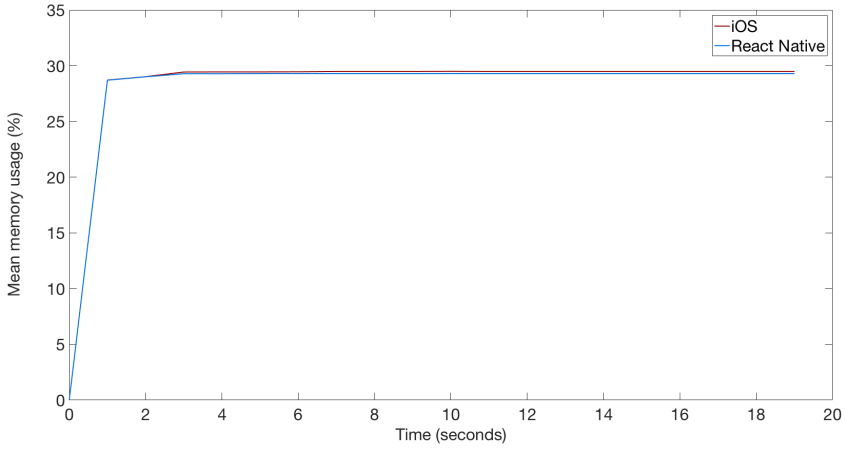

Figure 4.10: Mean % memory usage during scenario 1 for the React Native and iOS application.

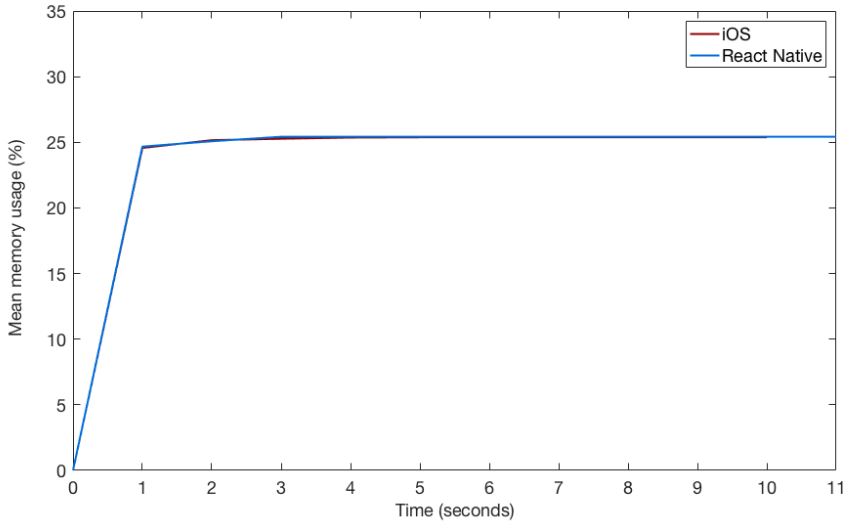

Figure 4.11: Mean % memory usage during scenario 2 for the React Native and iOS application.

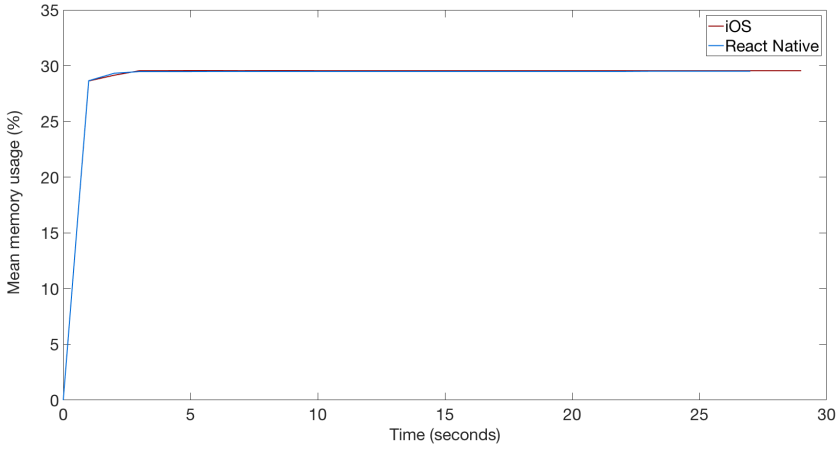

Figure 4.12: Mean % memory usage during scenario 3 for the React Native and iOS application.

4.2.3 Frames per second

This section presents the results of the FPS measurements that were performed during the scenarios, described in section 3.2.

The tables present the average number of dropped frames during the 5 test runs of each scenario. They also display the duration of the scenario and the resulting number of expected frames, given that the application should run at 60 FPS. Lastly, the tables display a percentage value based on the average number of frames dropped and the expected number of frames, to indicate the average drop rate.

The results shown in tables 4.1, 4.2 and 4.3 all indicate that React Native is equal or better than native Android.

Scenario 1	Native Android	React Native
Average frames dropped	12.2	7.4
Duration of scenario [s]	21	21
Expected number of frames	1260	1260
% average frames dropped	0.97%	0.59%

Table 4.1: Android FPS results from scenario 1.

Scenario 2	Native Android	React Native
Average frames dropped	16.6	11.4
Duration of scenario [s]	31	31
Expected number of frames	1860	1860
% average frames dropped	0.89%	0.61%

Table 4.2: Android FPS results from scenario 2.

Scenario 3	Native Android	React Native
Average frames dropped	12	5.6
Duration of scenario [s]	33	33
Expected number of frames	1980	1980
% average frames dropped	0.61%	0.28%

Table 4.3: Android FPS results from scenario 3.

4.2.4 Response time

This section presents the mean response times for the user interactions defined in table 3.1. Table 4.4 shows that React Native performs slightly worse for interaction 2, but better for all the others. The difference in response time for interaction 1, startup, is as large as 0.5 seconds.

The iOS results, presented in table 4.5, shows similar differences. React Native has faster response times for interactions 2,3 and 4 while iOS reigns supreme on interaction 1 and 5. The biggest difference is 636 ms for interaction 1, startup.

User interaction	Native Android [ms]	React Native [ms]
1	768.4	249.6
2	142.6	156.8
3	170.2	129.8
4	342.4	269.8
5	53.8	45.2

Table 4.4: Mean response times of user interactions on Android.

User interaction	Native iOS [ms]	React Native [ms]
1	1366	2002
2	464	310
3	430	376
4	80	42
5	404	428

Table 4.5: Mean response times of user interactions on iOS.

4.2.5 Application size

Tables 4.6 and 4.7 show the resulting application sizes, both installed application size and APK/IPA size.For both Android and iOS the React Native versions are larger, both in regards to installed application size and APK/IPA size.

Android	Native	React Native
Application size	13.39 MB	23.45 MB
APK size	2.4 MB	8.4 MB

Table 4.6: Application and APK size for both Android versions.

iOS	Native	React Native
Application size	12.21 MB	25.3 MB
IPA size	4.9 MB	13.8 MB

Table 4.7: Application and IPA size for both iOS versions.

4.3 Platform code sharing

Table 4.8 presents how many lines of code that could be shared between the Android and iOS applications using React Native. The table shows that 62% of all written code is shared between platforms and approximately 20% is specific for each platform. The Android-specific portion is slightly larger than the iOS-specific one.

	Shared	Android-specific	iOS-specific	Total
Lines of code	1308	433	359	2100
Percentage	62.29%	20.62%	17.10%	100.00%

Table 4.8: Platform code sharing of React Native applications.

4.4 Look and feel user study

This section present the results of the look and feel user study that was performed with the user tasks, described in section 3.4.1, and the standardized UEQ. The results from the Android platform is presented first, followed by the iOS results.

4.4.1 Android

Figure 4.13 displays the mean value for the native Android and the React Native application for each UEQ scale. Figure 4.13 also shows the confidence interval, displayed by the black range indicator that is emanating from the top of each bar. The mean score value as well as the confidence value of each bar in figure 4.13 are also presented numerically in table 4.9. When examining these values, it is clear that React Native have higher confidence values. This reflects that the spread is higher between the answers of the React Native testers.

The native Android application is given a higher mean UEQ score than the React Native version for all UEQ scales with the exception of Dependability. However, table 4.10 shows the t-test results and these conclude that there is no significant difference, using the α-value 0.05, between the results.

Table 4.11 shows Cronbach coefficients for each scale in the two Android applications. The values indicate how well the items of the scales are inter-

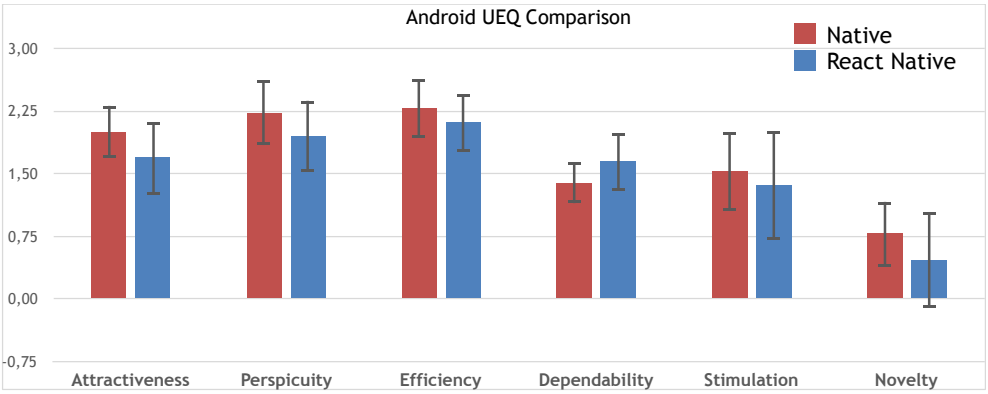

Figure 4.13: Android UEQ comparison with mean values for each scale.

	Native Android		React Native	
Scale	Mean	Confidence	Mean	Confidence
Attractiveness	2.00	0.31	1.69	0.43
Perspicuity	2.22	0.39	1.94	0.42
Efficiency	2.28	0.35	2.11	0.34
Dependability	1.39	0.25	1.64	0.34
Stimulation	1.53	0.47	1.36	0.65
Novelty	0.78	0.39	0.47	0.57

Table 4.9: Mean and confidence values for each scale from the Android UEQ.

preted by the user. Dependability and Novelty of the native application are exceptionally low, compared to the suggested, at least 0.6.

The differences in mean values can be observed in table 4.15, where a positive value means that the native application has a higher mean while a negative value indicate that the React Native application has a higher mean. The difference in mean value on the Android platform is on average 0.248.

UEQ Scale	T-test value	Significant difference?
Attractiveness	0.2599	No
Perspicuity	0.3562	No
Efficiency	0.5119	No
Dependability	0.2600	No
Stimulation	0.6899	No
Novelty	0.4016	No

Table 4.10: T-test results of the Android applications.

UEQ Scale	Native Android	React Native
Attractiveness	0.71	0.85
Perspicuity	0.77	0.87
Efficiency	0.61	0.55
Dependability	-0.87	0.66
Stimulation	0.69	0.88
Novelty	0.08	0.74

Table 4.11: Cronbach coefficients for each UEQ scale of the Android applications.

4.4.2 iOS

Figure 4.14 shows the mean values and confidence intervals of each UEQ scale for the iOS applications. These results are also presented numerically in table 4.12.

The native application has a higher mean value than the React Native version for three scales, Perspicuity, Stimulation and Novelty. React Native has the higher mean value for two of the scales, Efficiency and Dependability, while both applications have a mean value of 1.45 in regards to Attractiveness.

The t-test values presented in table 4.13 shows that there is no significant difference, using the α-value 0.05, for any of the scales.

Table 4.14 shows the Cronbach coefficient for each scale, using the previously mentioned α-value. It shows good values, compared to the suggested, at least 0.6, for every scale, except for native Efficiency and native Dependability where the values are poor.

The differences in mean values can be observed in table 4.15. The difference in mean value on the iOS platform is on average 0.257.

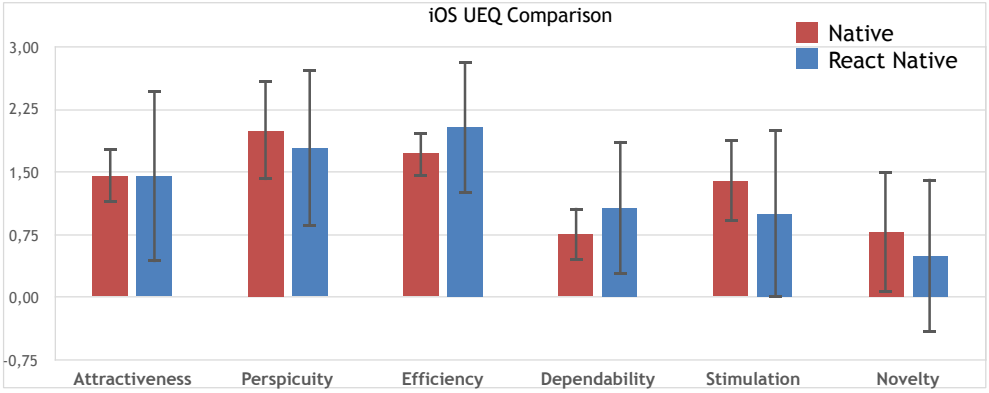

Figure 4.14: iOS UEQ comparison with mean values for each scale.

	Native iOS		React Native	
Scale	Mean	Confidence	Mean	Confidence
Attractiveness	1.45	0.32	1.45	1.03
Perspicuity	2.00	0.60	1.79	0.95
Efficiency	1.71	0.27	2.04	0.80
Dependability	0.75	0.32	1.07	0.80
Stimulation	1.39	0.49	1.00	1.01
Novelty	0.79	0.73	0.50	0.93

Table 4.12: Mean and confidence values for each scale from the iOS UEQ.

UEQ Scale	T-test value	Significant difference?
Attractiveness	1.0000	No
Perspicuity	0.7151	No
Efficiency	0.4770	No
Dependability	0.4851	No
Stimulation	0.5100	No
Novelty	0.6436	No

Table 4.13: T-test results for the iOS applications.

UEQ Scale	Native iOS	React Native
Attractiveness	0.68	0.94
Perspicuity	0.80	0.97
Efficiency	-0.20	0.89
Dependability	-0.34	0.80
Stimulation	0.55	0.93
Novelty	0.76	0.90

Table 4.14: Cronbach coefficients that show the correlation of items per scale for the iOS applications.

Scale	Android	iOS
Attractiveness	0.31	0.00
Perspicuity	0.28	0.21
Efficiency	0.17	-0.33
Dependability	-0.25	-0.32
Stimulation	0.17	0.39
Novelty	0.31	0.29

Table 4.15: Differences in mean values between native Android and React Native and between native iOS and React Native.

Chapter 5

Discussion

5.1 Results

Most of the collected results were slightly in favor of the native applications although React Native performed better than expected. The performance evaluation showed that the difference is small for this application. It was also shown that about 75% of the code is used in both versions of the React Native application. The look and feel was close to identical, as shown by the development results as well as the user study. The following sections present how the different results compare to the theory as well as to other studies.

5.1.1 Development

React Native is limited in the amount of available components compared to native development in Android and iOS. However, the framework does have an active community of developers that contribute with open source custom components. This extends the React Native base components to make up quite a large library. In React Native, plugins are easy to use and to get started with, however there are downsides too. The native source code of the plugins are of varying quality when anyone can contribute, this makes it necessary to sometimes customize the plugin. Again, depending on the quality of the plugin source code, customizing it can be either quick and easy or time consuming and intricate.

When developing the Android version of the React Native application there was no built-in slider component. However, it became available a few months later, with the release of React Native version 0.24 [1]. React Native is still a young framework and is updated frequently with new components.

[1] *React Native v0.23 Known Issues*, `http://facebook.github.io/react-native/releases/0.23/docs/known-issues.html#views`, Accessed: 2016-05-18

The Android support is not as extensive as the iOS one currently is, but it is rapidly catching up.

The application could have used more platform-specific features, for example the camera or Bluetooth. During the application specification, much focus was put on what features a standard application order to Attentec would include. If the focus was put elsewhere, this could have shown more weaknesses regarding platform-specific code and what features are available. It would also probably lower the amount of code that could be shared between platforms.

5.1.2 Performance evaluation

This section discusses how the performance results compare to existing theories and what causes the observed patterns and irregularities. This section also compares the results to other work, mostly [4], whose measurements are somewhat comparable to the measurements performed in this study.

CPU usage

The native Android application sends HTTP requests with every user input event. Even when native Android sends continuous HTTP requests in conjunction with the user input events it still consistently displays a lower CPU usage than React Native. Due to the application structure in the React Native version, where all data is requested at startup, no HTTP requests are sent to retrieve further data. The absence of significant differences in CPU usage at these events indicates that the HTTP requests does not significantly impact the CPU load.

The iOS applications expectedly displays an overall higher CPU consumption due to being run on a low-end device compared to the high-end Android device which also is the case in [4].

The React Native application needs to start its JavaScript thread in addition to the main thread. After that, it will create the virtual DOM using the JavaScript thread and send the resulting view to the main thread via the JavaScript bridge. This process seems to require extra CPU through the entire lifetime of the application and especially at start-up, which is clearly visible in the figures in section 4.2.1.

Scenario 2 ends with the opening of two graphs in the application. This is clearly indicated in the plot of React Native by two larger CPU spikes in figure 4.2. When looking at the native part, in figure 4.2, only one larger spike is visible, although it is displaced two time steps from the second large React Native spike. However, the collected data, which is used to construct the figures, indicates two larger spikes towards the end in native Android as well. Manually examining the collected data shows that the two native spikes has a mean value of 7% and 8%, which makes both of them only one percentage lower than their React Native counterpart. The first of these spikes disappears in the native plot because the spike has been displaced in

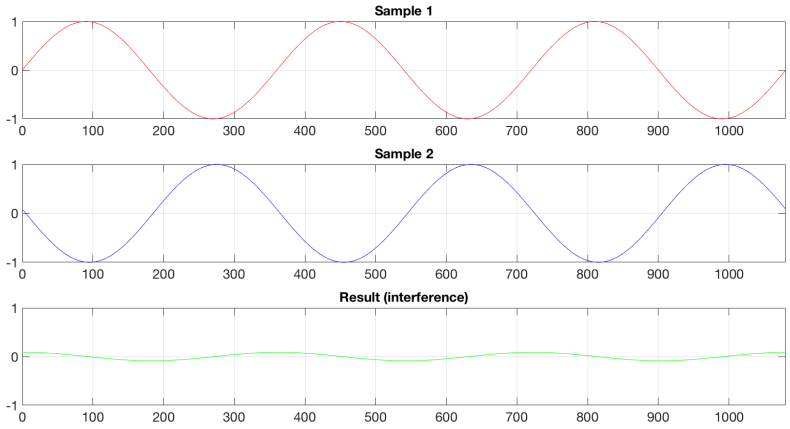

Figure 5.1: Interference effect when two sets of sample data have been added to each other.

some of the five readings. This means that when a mean value is calculated, the spike is spread out over an area with low values in between. This causes an interference pattern to occur, similar to the example in figure 5.1. In iOS the React Native data spikes are larger or equal to their native counterparts in all but one instance, the second spike in figure 4.6. This anomaly is also produced for the same reason as the missing native spike in figure 4.2.

Memory usage

The memory usage study shows promising results for React Native. The results are very similar for both platforms, React Native is close to equal to native in regards to memory usage.The iOS results show exactly the same results and while this seems promising it could actually mean that there is an issue with the selected measuring method. The method was however used by [4] and they achieved differing results between cross-platform and native on iOS. This indicates that either the results are valid, or that there has been a change in how the measurement tool works. No information about changes to the tool could be found and therefore we believe that the results are valid and reliable.

In regards to other Android work [4] shows that PhoneGap applications use significantly more memory than both native applications as well as applications that are built with Xamarin. Our results using React Native shows similar results to those using Xamarin on Android. This indicates that React Native could be similar to Xamarin in terms of memory usage.

If the iOS results are compared to those in [4] the memory usage of React Native is more similar to Xamarin than PhoneGap on iOS as well.

However, the results of their work shows that Xamarin uses more memory as the application is used, while the native version is relatively stable. Our results show that React Native is equal to the native version for the duration of all the scenarios.

Frames per second

Almost all of the frame drops happen when an animation is fired, for example, when a transition from the home-screen to the room-screen is initiated. The difference between the tests is that during these animations the native version always drops more frames.

Some of the animations were not performed in the same way on native and React Native and this may have impacted the results. More reliable results could have been obtained if more focus during development was put on using the exact same animations. If the same results were achieved with this approach, more conclusions could have been drawn regarding differences in the implementation of animations for native and React Native.

However, the drop rate is very low for all scenarios, with the average number of dropped frames never exceeding 1%. This means that even though differences can be detected between the native and the React Native application, they are small enough for it to not have any significant impact on the user experience.

Response time

The Android response time results, as seen in figure 4.4, showed that React Native is faster or equal to native in all test cases. The outlier is test number 1, application startup. In this test React Native is noticeably faster than the native version. This is probably due to implementation differences for the first render event. When you look at the applications you see that they take approximately the same amount of time to start. This result is positive for React Native and an important aspect as response times greatly affects the user experience.

Figure 4.5 shows the iOS response time results. In this case, it is also the application startup time that stands out, but this time the native version is faster. This is probably an effect of having to start the separate Javascript environment to run React Native code. It is however peculiar that this shows up in the iOS tests, but not in the Android tests. In all other tests, the React Native response times are faster or equal to the response times of the native version.

On both platforms we can see that all interactions after startup has a response time that is shorter than 1 second. This is important to make sure that the user does not lose interest in the application [11]. This result was expected for the native versions but uncertain for React Native. The result is very beneficial for React Native as it confirms its usefulness in producing an application with good user experience in regards to speed.

Application size

The application size results came out as expected. Since the React Native applications requires the framework to be part of the application these versions were bigger, both in regards to APK/IPA size and installed application size. The same results were achieved by [4], but for the mobile frameworks Xamarin and PhoneGap. The overhead size of the framework seems to be about 10-15 MB for installed applications and 5-10 MB for the APK/IPA.

To be able to make further conclusions about how the application size changes depending on the application, different tests needs to be conducted. If several different native and React Native applications were developed, the sizes of these applications could be compared and definite conclusions regarding the size of the overhead could be made.

5.1.3 Platform code sharing

The resulting platform code sharing shows that over 60% of written code could be used for both Android and iOS. Facebook claims that they could share 85% of the code for the first React Native Android application [8]. This rather large difference in sharable code compared to our result can be explained by the different approaches used when building the cross-platform applications. Their approach was to port an existing React Native iOS-application to Android and therefore their measurements are made on how much code that can be shared to an Android application from an all iOS React Native application. If our application is treated as two separate entities in the same manner, the results show that 78% of the iOS-application could be used in Android and 75% of the Android-application could be used in iOS.

The results also show that, for this application, about the same amount of platform-specific code was needed for each platform. This equality is fairly surprising as no compromises were made regarding UI design to fit both platforms and make each platform look and feel native.

The high amount of sharable code presented in the results are satisfiable and deemed as good for a cross platform framework as long as the application also fulfills its other requirements regarding performance and user experience.

5.1.4 Look and feel user study

The look and feel user study results show that the applications have very similar user experiences, as we expected after the development phase. The differences shown in table 4.15 are quite low and even the maximum value of 0.39 is not a very significant difference. Given the low number of participants and the high confidence values observed in tables 4.9 and 4.12, the mean results are still quite uncertain. There is no way of telling if the trends of the

gathered data will keep occurring with more respondents or if the already gathered data are outliers.

The confidence intervals are much higher for participants in the React Native on iOS category than for the rest. This is believed to be an effect of the different respondents that answered the questionnaire. Due to randomness the respondents of the React Native survey had different views on technology and how they approached the application. The total iOS confidence intervals are likely larger due to the lower number of respondents for this platform.

The t-test results show that there are no significant differences between the native versions and their React Native counterparts. A result with a significant difference can be said to be attributed to actual differences and thereby not randomness with a certainty of 95%. On the other hand, when there are no significant differences the outcome is more likely a result of chance. The result can of course be that there actually are no differences, however more respondents are required to statistically justify such a result and to increase its reliability. [22]

There were some issues where users interpreted related questions differently. For example, question 17 in the UEQ asks if the application is secure or not secure. This relates to Dependability and should be interpreted the same way as question 8, which asks if the application is unpredictable or predictable. Many of our respondents interpreted question 17 as how secure the data transfers are, which is hard to answer. However, this showed in the Cronbach coefficient values for each scale and was therefore taken into account. The Novelty scale was also open to interpretation since this depends on how experienced the user is with technology. It also depends on what a user treats as new and innovative. Furthermore, we did not try to create something that was innovative, but rather something that followed design guidelines and conventions. The Novelty score is therefore the scale with the lowest mean value, as expected.

The Cronbach coefficient values are extremely low for dependability and novelty on the native Android application, which means that the mean result is unreliable until more people answer the questionnaire. The low Cronbach coefficients can be tracked back to single respondents that interpreted two questions related to each scale very differently. Similarly, the Efficiency, Dependability and Stimulation scales for native iOS also displayed low Cronbach values. Since there were only 18 respondents on Android and 14 on iOS, misinterpretations of questions have had a huge impact on the results. All the other scales have good Cronbach coefficients in accordance with the definition in section 2.4.2, which shows that most respondents have interpreted the questions the same way and provided similar answers to related questions. The scales with low Cronbach values was interpreted carefully as we did not have enough respondents to judge if the questionnaire needed changing or if it was just a coincidence that people misinterpreted the questions.

In regards to other work, [12] have completed a similar study where they compared Titanium to native versions of an application. They chose to calculate the mean UEQ score for both versions of Titanium and comparing with the mean for both native applications. The largest difference they found were the Attractiveness of the applications, where native scored 2.234 and Titanium scored 1.59. If we calculate the average mean between our Android and iOS applications for the Attractiveness scale we attain a score of 1.57 for React Native and 1.725 for native. The difference between native and React Native is smaller than the difference between native and Titanium by quite a large margin. Since Titanium uses the same approach as React Native, native scripting, this is quite an interesting result. However, different applications were used and none of the studies have a lot of respondents. The study performed in [12] was performed in 2014, which means that Titanium can have improved to the point where it is equal to React Native in 2016. It is however notable that the average difference in mean score for each scale clearly favors React Native over Titanium for these studies.

5.2 Method

This section presents a critical review of the methods used in this report. Each section is evaluated in regards to replicability, reliability and validity.

5.2.1 Development

The development phase included creating an application concept, thereafter creating UI concept images and from this, create a feature backlog. If the feature backlog had been created directly, the time consumption of the development phase could have been reduced. However, it would have been harder to assure that native guidelines would have been followed during development. With more than one developer and without a clear and common view of the application of the there would have been a need for a lot more interruptions to discuss styling issues. These extra interruptions are believed to have been a lot more time consuming than the initial work with the UI concepts. In the end, when creating a common foundation on which to build the applications we were most likely able to save ourselves a lot of time and hazel further down the road. It made it easier to stay synchronized and to be sure that we were working towards the same goal.

Simultaneously, the UI concept that was created was not overly clear on the details of the application such as font size, width and height of fields, coloring and so forth. These minor details were later the cause for some discussions that most likely slowed down the development process significantly. In hindsight, even more time and work could have been put into the UI concept to increase the aforementioned advantages of having a UI concept even more.

Kanban was used as a management system to know what features each person was working on at any moment. Kanban was used to ensure that an agile process was followed, since this has shown good results for previous studies. It was also important to be flexible when we developed several different applications. If Scrum had been used, a lot of extra overhead work would have been necessary, such as using planning poker, and this would have lowered effective development time. Since each application only had about 3 weeks of development time, it would have been hard to split the development into sprints and with only one sprint, Scrum is hardly worthwhile. Further, with only two developers, morning meetings did not seem as an effective approach, but rather asking questions as soon as they came up to quickly resolve issues and keep working. With Kanban, it was also easy to see what features each person was working on using the visual board system with notes.

5.2.2 Performance evaluation

The collection of CPU usage, memory usage and FPS data was handled differently on each platform. For Android, the retrieval of data was reliable for sampling once every second, but for smaller sample times the results would sometimes include more or less sampling points than expected. For iOS, the sampled data could not be exported, but had to be copied by hand. Since the scenarios are between 20-30 seconds long, it would not have been feasible to use a smaller sample time than once every second. If this was not the case, a smaller sampling time would have been preferable to get more reliable measurements.

CPU usage

When processing the collected datam a mean value for each sampling point was calculated and this was presented as a graph in the results. This method of compiling the five sets of sample data for each configuration might have been misleading. Closer inspection of the data showed that some of the sample sets were displaced from others by one or two time steps. This displacement often caused an interference pattern between the samples when they were added together to calculate the mean value. Instead of using a mean value to summarize the sample sets, the median value might have been better suited. If there was enough time, a manual inspection of the data to determine one sample set that best represents the average behavior might have been an even more preferable choice.

The three scenarios that were used are all fairly slow moving and based on regular user behavior. This improved the validity of the tests however, it also resulted in a lower than expected overall CPU usage on both platforms. Even though React Native clearly uses more CPU than native versions, all of the applications display a low CPU load and currently the difference is too small to impact the user experience. However, a more CPU intensive

scenario or changes to the application could have altered this result. With a higher overall CPU load we believe that the difference between the versions would have been more noticeable and that the use of React Native could have had a negative impact.

Memory usage

It is possible that the maximum memory that was used is higher than what is shown in our data, due to larger sample data hiding in between the sample points. However, this is probably not an issue, since the possible spikes are short and average memory consumption is small in comparison to the total available memory.

The automated scenarios were specifically used to both increase replicability and reliability and to reflect realistic use cases. They were also chosen to be exhaustive and evaluate parts of the application that were suspected to be performance-intensive. The scenarios were designed with the target application in mind, which makes it hard to compare the results to the results of performance tests conducted by other cross-platform works. Therefore, the scenarios could have been made shorter or longer in order to match the scenarios of other work. However, by doing so the benefits of customizing the scenarios to the application would most likely have been wasted.

As shown by [4] the memory allocation handling is implemented differently depending on the mobile device. Therefore, it could have been valuable to run the performance tests on different devices in order to conclude more about the impact that React Native have on memory consumption.

Frames per second

Using Systrace to measure UI performance is an approach that both Google and Facebook recommends [2] [3]. It supplies traces that include every single method call and event that the application handles. It is also very easy to see exactly where frames were dropped and how many they were. This gives a high level of validity to the results produced by Systrace.

The recommended approach for UI testing on iOS was to use Instruments. This approach was tried, but did not provide any usable data. It could provide measurements once every second showing how many frames that were rendered the previous second. A problem with this was that when the UI was not updated, this was reported as 0 frames. It was also not exportable and hard to automate. For these reasons, the reliability of the produced data was considered too low to be included in this report.

The scenarios included all of the available animations in the application, which allowed for complete FPS data for the entire application to be

[2] *Analyzing UI Performance with Systrace*, `https://developer.android.com/tools/debugging/systrace.html`, Accessed: 2016-02-15

[3] *React Native Android UI Performance*, `https://facebook.github.io/react-native/docs/android-ui-performance.html#content`, Accessed: 2016-05-02

collected and analyzed.

Response time

The response time measurements were performed in vastly different ways on Android and iOS. For Android, measurements were taken between the user input event and the succeeding render event. For iOS, a similar approach was tested however that did not yield reliable results because of limitations in Instruments. Instead, video recordings of the user interactions were taken and examined. Since the aim of this work is not to compare response times between native platforms, but rather between versions of the application on the same platform, using two different methods was deemed to be the best approach.

The approach of using video recordings could have been used for Android as well, to compare the results, but we felt this introduced more possible error sources. For example, the time of the input event would not be recognized by the OS, but rather the time of the visual event when a button is pressed would be used. We could also have tried to use logging the time of different activity lifecycle events to deduce the response times from this instead of using the Systrace traces [4]. However, since React Native does not use the same lifecycle events as native Android, we felt that this would not be a fair measurement.

For iOS, no reliable way to export a trace containing user input events and corresponding render events could be found. Therefore, event driven logging was performed to acquire Unix time stamps for each event. This approach proved difficult as the results were far from what was expected and they were deemed improbable. For instance, the native response times were very fast, some as low as 0.3 ms which led us to conclude that this approach had very low validity. Instead, we decided to use video recordings, which unfortunately introduced new error sources. For example, the actual time that the device receives the input event is not logged but rather the time of a visual event, visibility on the screen. Furthermore, during the examination of the visual events in the video recordings, the precision was limited to the frame rate of 60 FPS in the video. This means that in the worst case, the result of a response time can have a error margin of 16 ms.

Application size

The application size was measured in two ways, APK/IPA size as well as installed application size. The APK/IPA size was measured on files built from the platform-specific tools. However, when a user uses these installers, they are most often downloaded from an app store. These app stores often add more overhead to the files, especially for iOS. To upload an application to the Apple app store, you need to submit the application for review. This is a time- and resource-expensive venture and therefore this was discarded. Instead, the approach of building the installers and measuring their size was

chosen. The expected result is that the store owner would add an equal amount of overhead to both versions of the application, since they are very similar.

The installed application size was measured using the integrated OS tools. This method was chosen because these tools were developed by iOS- and Android-developers who are considered a trustworthy source. The results are considered to have a high validity and to be the most accurate results available.

5.2.3 Platform code sharing

The way that platform code sharing was measured has some drawbacks. For example it relies on the fact that each line can be classified as shared or platform-specific. This was made easier as React Native allows platform specific file extensions. Each file that ended with ".android.js" was counted as Android-specific and each file that ended with ".ios.js" was counted as iOS-specific.

Plugins were also hard to take into account, since they are not written in Javascript, nor written by us. We decided to exclude these plugins from the classification entirely. The plugins that are available are often iOS-exclusive. Due to this, it is often much easier to build an iOS-application where you write fewer lines of code. However, you may end up using more code if plugins are included. For our application, as few plugins as possible were used and those that were used were made sure to be available for both platforms.

Since we could not find any automated way of classifying the code, there was really no other option but manual classification. We did not compare any code between native and React Native, since they are written in different languages and it would not be possible to draw reliable conclusions from it.

5.2.4 Look and feel user study

The look and feel user study was conducted through a standardized user experience questionnaire. It is possible that we could have achieved a more thorough analysis of the look and feel if we, for example, used a thinking out loud-protocol where we tried to analyze how the user felt in combination with a questionnaire. However, since that approach is more time consuming we felt that it was not feasible to conduct such a user study within the limited time frame. It could also have been interesting to use a non-standardized questionnaire, where users can describe how they felt using the application. This would make it harder to analyze the data, since it would need some classification of results as well. We also felt that it would be harder to compare the applications to each other if the results were classified by us.

The UEQ approach also made it easy to conduct the study without having to distribute the application to other people, since we were always present during the tests. Another approach would have been a longitudinal

test, where the users used the applications for a longer duration and then gave their opinions. We felt that we wanted to test the users immediate feelings, even though it is possible that users would have a different opinion after a longitudinal test. The longitudinal test would also not be controlled by our given user tasks, but rather how a user would use the application. This could make it harder to replicate the study, since every user would likely have a different experience of the application. Since we were aiming to make the study replicable this was another reason to choose the standardized approach.

We chose not to modify the UEQ, even if the handbook permits it [13]. We could have excluded a scale, for example novelty, that was not really interesting in regards to our developed application. However, since we were only trying to compare the two applications, that were supposed to be identical, we felt that they should be equal in all respects. There was also the possibility of changing the wording of a question. The question we had the most issues with was question 17, not secure or secure. If we did this study again we would probably change the wording of this question. Several users said this was the question where they did not know where to answer and this led to a low correlation and Cronbach coefficient for the dependability scale.

Regarding the execution of the user tests we also considered letting each user test both versions for a specific OS. This would give us more results to compare without conducting more user tests. However, we would then have to consider the fact that some users would use the native version first and some users would use the React Native version first. These differences would have to be investigated to make sure that there was no bias towards either version. We also tried doing such a test for an iOS user, who felt that the test was hard and tried to remember how the first version was evaluated during the evaluation of the second application, since they were so similar. Because of this, we felt that we would get more honest results using our selected approach. This latter approach would also make the test twice as time consuming and that could deter user candidates.

5.2.5 References

When collecting information from research papers that contain related work as well as information about how to go about when performing a user experience survey, we aimed to find high quality, original publications. A publication was considered to be of a high quality when it showed a high level of replicability, reliability and validity. We also made sure that the publications and articles used did not contain outdated information and that they conveyed a hallmark of professionalism. Furthermore, first hand references were sought after in order for the retrieved information to be complete and not altered too much in a chain of rewritings. However, since the number of publications is small for cross-platform framework evaluations, we had to

use the best available. We always evaluated the publications with a critical attitude.

For purely technical information regarding frameworks, programming languages, mobile platforms and other techniques we also aimed to use first hand references to the greatest extent possible. This often meant using the documentation for the techniques, which we considered to be reliable sources of information.

React Native is the main subject of this thesis work and like other frameworks, we turned to the documentation of the framework to gather information about it. However, in order to acquire a deeper understanding about the framework we needed to go beyond the documentation. React Native is a young framework and consequently not much research has yet been made on the subject. This forced us to turn to less conventional sources of information, such as key note speeches and blog articles. When we used these kinds of sources we were well aware of the authors affiliation with Facebook who created the React Native framework. When examining the content of these references, a critical attitude was assumed in order to screen the information for factual data.

5.3 Broader perspective

Mobile applications have become a central part of life and people use them daily and for many day-to-day activities. Keeping mobile applications and the development of these readily accessible is therefore of great importance to society. Buying native applications for more than one platform from professionals are often considered expensive which greatly limits the number of potential customers for a software development company. A cross-platform framework, such as React Native, is able to reduce the development time and thus the cost of mobile applications that are suitable for both Android and iOS devices.

If a software development company should consider React Native applications sufficiently good compared to native applications, they could choose to not develop native applications at all. The company might previously have one employee developing native Android and one employee developing native iOS applications. The immediate concern for switching to React Native and by that discarding native development is whether or not there will be enough work to be made for the two previous native application developers to both still have a job after the transition. This reasoning can be taken one step further to include web developers as well, since React Native is actually built upon the web JavaScript framework ReactJS. In theory, because of the similarities between React Native and ReactJS, this means that both of the native app developers could be fired to be replaced by a web developer. The web developer should be able to translate his or her ReactJS knowledge to start developing mobile applications as well.

React Native is an open source framework that is developed by Facebook.

An open source framework is advantageous from an ethical point of view, as it does not allow for the code to be used for any ulterior motive without the community discovering this. Apart from this, developers and users will never become dependent on a company for their applications to run. They can always trust that information that is sent through their application is not disclosed or used in a way that is insulting one's integrity.

Chapter 6

Conclusion

The goal of this thesis was to evaluate React Native and how it can be an option for mobile application cross-platform development. One application was developed using React Native as well as one native version for each platform. The React Native application uses two different UI-configurations to look and feel native. These versions were then compared to their native counterparts.

The results were promising for React Native and no critical performance issues could be found. We recommend that future work on React Native performance focuses on possible issues regarding CPU usage. Even with this low intensity application, the CPU load became surprisingly high during user interactions and the difference from native could become troubling for more CPU-intensive applications. In regards to application size, it was always larger for React Native, which was the expected result. This thesis work was not able to conclude if the application size overhead is constant or if it is affected by different application features. In regards to the other performance metrics no concerning differences could be found. Instead, the results were surprisingly good, since comparisons for other frameworks with native versions has shown negative results for the cross-platform frameworks.

The results regarding cross-platform code sharing shows similar results to what Facebook claimed about their first Android-ported application. About 75% of the application can be used for both the Android and iOS applications, which is great both from a development time perspective as well as a maintainability perspective. However, as the developed application only uses platform-specific code for user interface configurations, it would be interesting to see how this number is affected if more platform-specific features, e.g. camera or Bluetooth, were added.

It is difficult to draw conclusions from the look and feel study, due to the small number of respondents. We can definitely say that our results points toward equal user experiences, since no significant differences could be found in the results of the UEQ. However, more respondents would make

the results more reliable and slight differences could then have been detected and statistically confirmed. More extensive user studies are required to map the effects that React Native has on the user experience. We also believe that performance issues would affect the user experience in such a way that, if there were any, the issues would be found through an extensive user study.

In total we would recommend using React Native for cross-platform development. However, some application are more suitable for React Native than others as integrated mobile features such as Bluetooth are not available without extensive workarounds. With the help of this technology, companies should be able to develop applications for several platforms faster, cheaper and with close to the same results as native development. This could lead to more business for a company and in turn more available work for developers. It could also lead to cheaper mobile applications for customers that want their application to support several platforms.

Bibliography

[1] P. Hunt, "React: Rethinking best practices." JSConf EU, 2013.

[2] T. Occhino, "Keynote." React.js Conf, 2015.

[3] T. Occhino, "React native: Bringing modern web techniques to mobile," March 2015. Accessed: 2016-02-08.

[4] D. Spinellis, "Recruiting a Star Team," *IEEE Software*, vol. 32, no. 3, pp. 3–5, 2015.

[5] C. P. Rahul Raj and S. B. Tolety, "A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach," *2012 Annual IEEE India Conference, INDICON 2012*, pp. 625–629, 2012.

[6] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, "Evaluating Cross-Platform Development Approaches for Mobile Applications,"

[7] I. Dalmasso, S. K. Datta, C. Bonnet, and N. Nikaein, "Survey, comparison and evaluation of cross platform mobile application development tools," *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 323–328, 2013.

[8] P. von Weitershausen and D. Witte, "React native for android: How we built the first cross-platform react native app." Accessed: 2015-12-02, September 2015.

[9] Google, "Android stack figure." `https://source.android.com/images/android_framework_details.png`. Accessed: 2016-02-15, shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

[10] Google, "Android activity lifecycle figure." `http://developer.android.com/images/activity_lifecycle.png`. Accessed: 2016-02-15, shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

[11] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[12] E. Angulo and X. Ferre, "A Case Study on Cross-Platform Development Frameworks for Mobile Applications and UX," *Proceedings of the XV International Conference on Human Computer Interaction - Interacción '14*, pp. 1–8, 2014.

[13] M. Schrepp, "UEQ - User Experience Questionnaire Handbook," pp. 1–11, 2015.

[14] "Construction and Evaluation of a User Experience Questionnaire," *HCI and Usability for Education and Work*, pp. 63–76, 2008.

[15] S. Pathologist, "This document : More Project information and further documents : SUS - A quick and dirty usability scale," pp. 1–8.

[16] E. Angulo, "UX & Cross-Platform Mobile Application Development Frameworks," 2014.

[17] S. R. Humayoun, S. Ehrhart, and A. Ebert, "Developing mobile apps using cross-platform frameworks: A case study," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8004 LNCS, no. PART 1, pp. 371–380, 2013.

[18] P. R.M.de Andrade, A. B.Albuquerque, O. F. Frota, R. V Silveira, and F. A. da Silva, "Cross Platform App : A Comparative Study," *International Journal of Computer Science and Information Technology*, vol. 7, no. 1, pp. 33–40, 2015.

[19] M. Ciman and O. Gaggi, "Evaluating impact of cross-platform frameworks in energy consumption of mobile applications," *WEBIST14 - 10th International Conference on Web Information Systems and Technologies*, 2014.

[20] M. Ng Moon Hui ; Liu Ban Chieng ; Wen Yin Ting ; Mohamed, H.H. ; Rafie Hj Mohd Arshad, "Cross-platform mobile applications for android and ios," in *Wireless and Mobile Networking Conference (WMNC), 2013 6th Joint IFIP*, pp. 1–4.

[21] D. J. Anderson, *Kanban*. Blue Hole Press, 2010.

[22] D. J. P. Biddix, "Research rundowns, significance testing (t-test)." Accessed: 1016-06-01, 2009-07-17.

Appendices

Appendix A

UI Concept

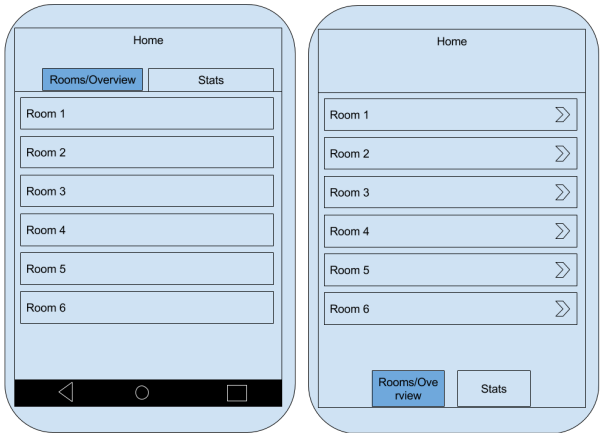

Figure A.1: The Android and iOS Home Screen concepts.

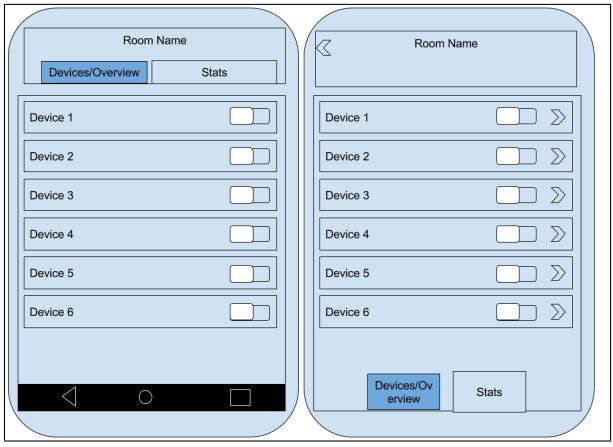

Figure A.2: The Android and iOS Room Screen concepts.

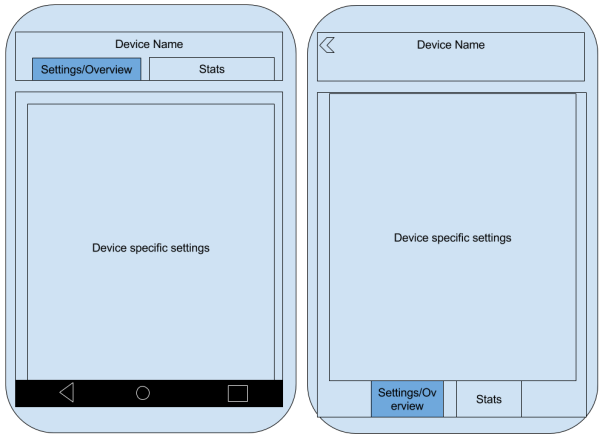

Figure A.3: The Android and iOS Device Screen concepts.

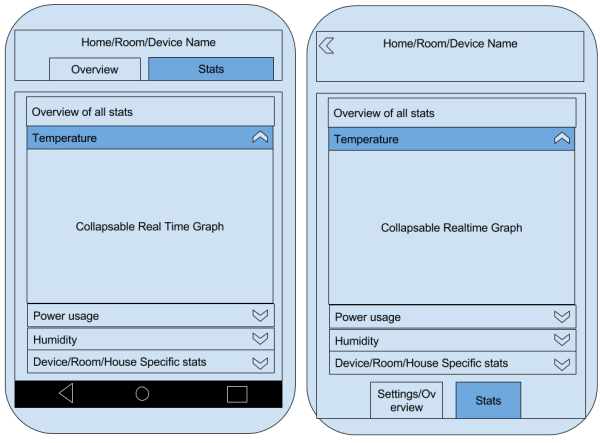

Figure A.4: The Android and iOS Stats Screen concepts.

Appendix B

Development Results

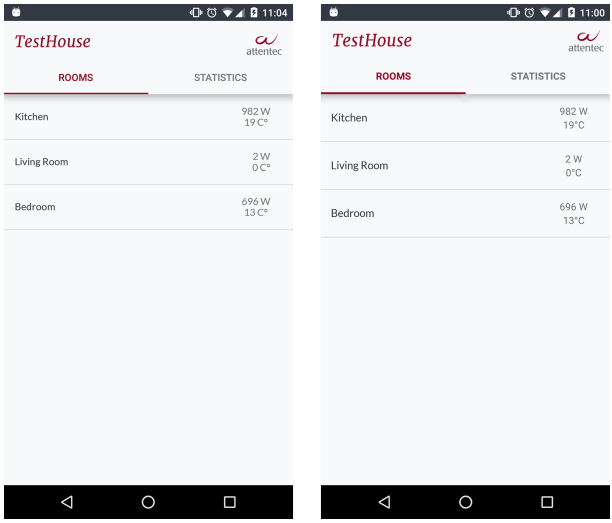

Figure B.1: The Android home screen results, native to the left and React Native to the right.

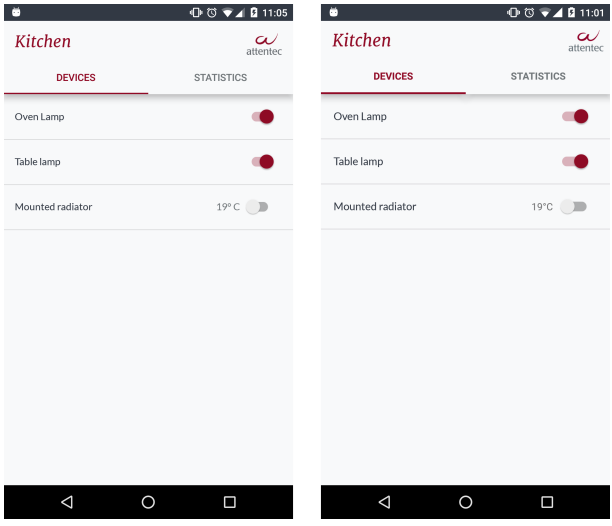

Figure B.2: The Android room screen results, native to the left and React Native to the right.

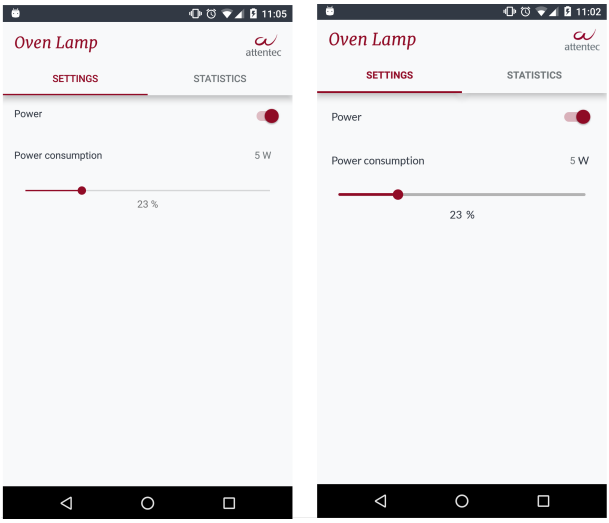

Figure B.3: The Android device screen results, native to the left and React Native to the right.

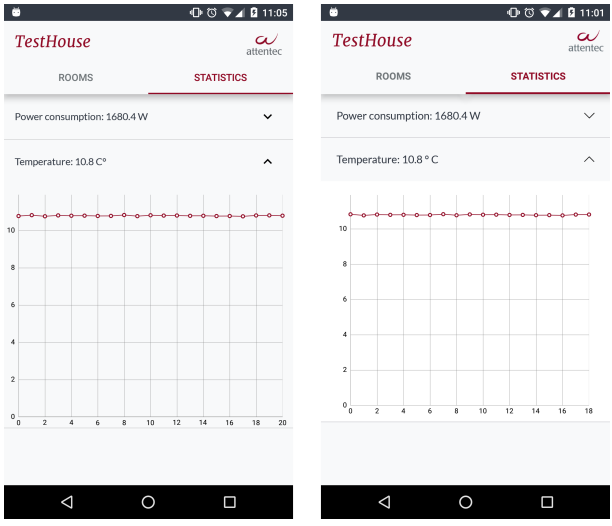

Figure B.4: The Android stats screen results, native to the left and React Native to the right.

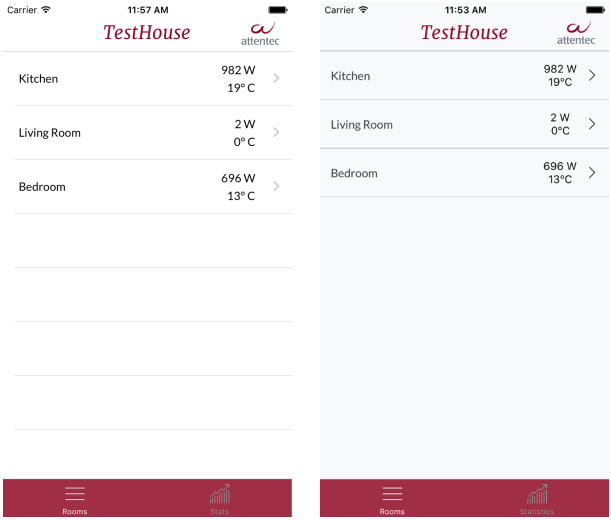

Figure B.5: The iOS home screen results, native to the left and React Native to the right.

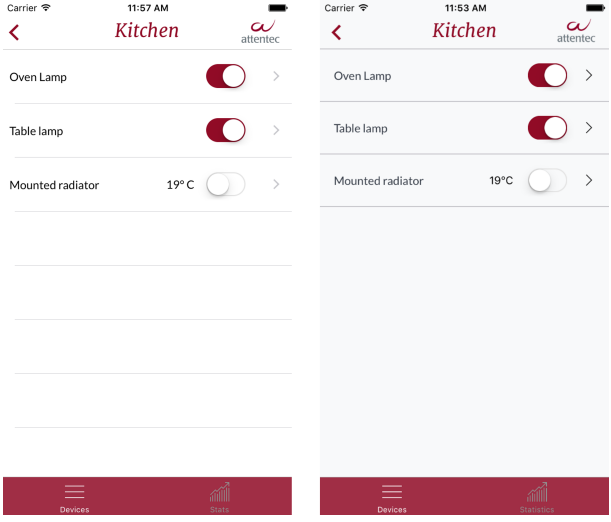

Figure B.6: The iOS room screen results, native to the left and React Native to the right.

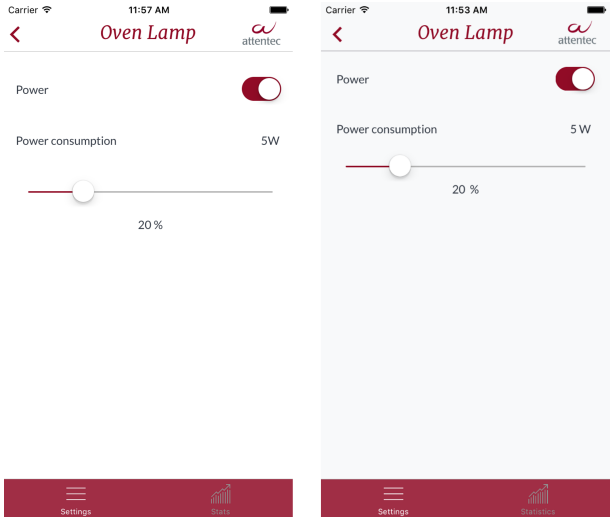

Figure B.7: The iOS device screen results, native to the left and React Native to the right.

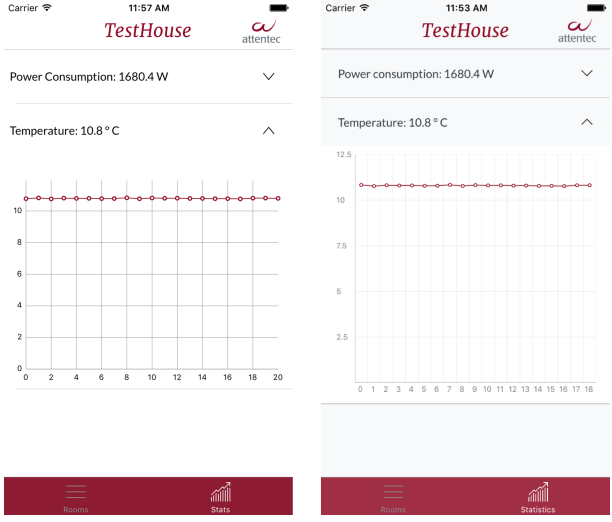

Figure B.8: The iOS stats screen results, native to the left and React Native to the right.

Appendix C

Feature backlog

C.1 Mobile application

The tasks for the mobile application which will be developed for iOS and Android using native techniques as well as React Native.

As an application user I want to be able to:

A.1 view a home-screen.

A.2 view a list of all rooms in a house.

A.3 view a room-screen.

A.4 view a list of all devices in a room.

A.5 view a device-screen.

A.6 see info and settings about a lamp.

A.7 turn on and off a device on the device-screen.

A.8 view power consumption of a device.

A.9 dim the light of a lamp.

A.10 turn on and off a device on the room-screen.

A.11 view a stats-screen about a single room.

A.12 view a stats-screen about a single device.

A.13 view a stats-screen about the entire house.

A.14 view power consumption of a house.

A.15 view power consumption of a room.

A.16 view a line graph.

A.17 see info and settings about a radiator.

A.18 change the temperature of a radiator.

A.19 see temperature of a radiator on the room-screen.

A.20 view temperature of the house.

A.21 view temperature of a room.

A.22 view a line graph that updates in real time.

C.2 Back end

The tasks for the back end which will be developed as a RESTful API using Express.js, MongoDB, Mongoose ORM.

As a developer I want to be able to

B.1 communicate with the application using a RESTful API using HTTP Requests.

B.2 HTTP GET/POST/PUT/DELETE model data for a house.

B.3 HTTP GET/POST/PUT/DELETE model data for a room.

B.4 HTTP GET/POST/PUT/DELETE model data for a lamp device.

B.5 HTTP GET/POST/PUT/DELETE model data for a radiator device.

Appendix D

Test scenarios

This appendix contains the automated test scenarios used to evaluate the performance of the applications.

D.1 Android

androidScenarios.py **Page 1**

```
#! /usr/bin/env python
# −*− coding: utf−8 −*−

import sys, os, argparse, re
from datetime import datetime
from com.dtmilano.android.viewclient import ViewClient

_s = 1
def main(argv):
    package = "se.attentec.attenhome"
    parser = argparse.ArgumentParser(description='Start Android scenarios')
    parser.add_argument('−rn', help="sets react as the platform", action="store_true
")
    parser.add_argument('−android', help="sets android as the platform", action="sto
re_true")
    parser.add_argument('−t', type=int, nargs='?', help="sets the test to be run [1.
..8]", choices=xrange(1,9), required=True)
    args = vars(parser.parse_args())
    if args['rn']:
        package = "com.attenhome"
    if args['t'] == 1:
        #EXPAND BOTH GRAPHS HOME SCREEN
        startApplication(package)
        expandHomeStats()
    elif args['t'] == 2:
        #EXPAND BOTH GRAPHS RADIATOR SCREEN
        startApplication(package)
        expandRadiatorStats()
    elif args['t'] == 3:
        #SELECT A DEVICE FLIP SWITCH, BACK FLIP SWITCH AGAIN
        startApplication(package)
        selectDeviceFlipSwitches(args['rn'])
    elif args['t'] == 4:
        #APP STARTUP
        startApplication(package)
    elif args['t'] == 5:
        #SELECT ROOM
        selectRoom()
    elif args['t'] == 6:
        #SELECT DEVICE
        selectDevice()
    elif args['t'] == 7:
        #SWITCH TO STATS TAB HOME SCREEN
        goToHomeStats()
    elif args['t'] == 8:
        #GO BACK FROM DEVICE SCREEN
        backFromDevice()
    else:
        print 'Test number was not valid (needs to be in [1...8])'
    platform = 'React Native' if args['rn'] else 'Android'
    print 'Test number ' + str(args['t']) + ' was completed on ' + platform + ' at '
 + str(datetime.now().time())

def expandHomeStats():
    vc.dump(window=−1)
    vc.findViewWithTextOrRaise(re.compile('Statistics|STATISTICS')).touch()
    print("Pressed Stats tab at " + str(datetime.now().time()))
    vc.sleep(_s)

    vc.dump(window=−1)
    vc.findViewWithTextOrRaise(re.compile('Power consumption*')).touch()
    print("Expanded power consumption graph at " + str(datetime.now().time()))
    vc.sleep(_s)

    vc.dump(window=−1)
    vc.findViewWithTextOrRaise(re.compile('Temperature*')).touch()
    print("Expanded temperature graph at " + str(datetime.now().time()))
    vc.dump(window=−1)
    vc.sleep(_s)

def expandRadiatorStats():
    vc.dump(window=−1)
    vc.findViewWithTextOrRaise(u'Kitchen').touch()
```

```
    print("Pressed kitchen at " + str(datetime.now().time()))
    vc.sleep(_s)

    vc.dump(window=-1)
    vc.findViewWithTextOrRaise(u'Mounted radiator').touch()
    print("Pressed Mounted Radiator at " + str(datetime.now().time()))
    vc.sleep(_s)

    vc.dump(window=-1)
    vc.findViewWithTextOrRaise(re.compile('Statistics|STATISTICS')).touch()
    print("Pressed Stats tab at " + str(datetime.now().time()))
    vc.sleep(_s)

    vc.dump(window=-1)
    vc.findViewWithTextOrRaise(re.compile('Power consumption*')).touch()
    print("Expanded power consumption graph at " + str(datetime.now().time()))
    vc.sleep(_s)

    vc.dump(window=-1)
    vc.findViewWithTextOrRaise(re.compile('Temperature*')).touch()
    print("Expanded temperature graph at " + str(datetime.now().time()))
    vc.dump(window=-1)
    vc.sleep(_s)

def selectDeviceFlipSwitches(isReact):
    vc.dump(window=-1)
    vc.findViewWithTextOrRaise(u'Kitchen').touch()
    print("Pressed kitchen at " + str(datetime.now().time()))
    vc.sleep(_s)

    vc.dump(window=-1)
    vc.findViewWithTextOrRaise(u'Oven Lamp').touch()
    print("Pressed oven lamp at " + str(datetime.now().time()))
    vc.sleep(_s)

    vc.dump(window=-1)
    if isReact:
        switch = vc.findViewById('id/no_id/9')
    else:
        switch = vc.findViewById('se.attentec.attenhome:id/powerswitch')
    switch.touch()
    print("Flipped switch on device screen at " + str(datetime.now().time()))
    vc.sleep(_s)

    vc.dump(window=-1)
    device.press('KEYCODE_BACK')
    print("Pressed Back at " + str(datetime.now().time()))
    vc.sleep(_s)

    vc.dump(window=-1)
    if isReact:
        switch = vc.findViewById('id/no_id/11')
    else:
        switch = vc.findViewById('se.attentec.attenhome:id/list_power_switch')
    switch.touch()
    print("Flipped switch on room screen at " + str(datetime.now().time()))
    vc.dump(window=-1)
    vc.sleep(_s)

def startApplication(package):
    activity = '.MainActivity'
    component = package + "/" + activity
    #device.shell('am force-stop ' + package)
    device.startActivity(component=component)
    print("Started application at " + str(datetime.now().time()))
    vc.dump(window=-1)


def selectRoom():
    vc.dump(window=-1)
    vc.findViewWithTextOrRaise(u'Kitchen').touch()
    print("Pressed kitchen at " + str(datetime.now().time()))
    vc.dump(window=-1)
```

```
    vc.sleep(_s)

def selectDevice():
    vc.dump(window=-1)
    vc.findViewWithTextOrRaise(u'Oven Lamp').touch()
    print("Pressed oven lamp at " + str(datetime.now().time()))
    vc.dump(window=-1)
    vc.sleep(_s)

def goToHomeStats():
    vc.dump(window=-1)
    vc.findViewWithTextOrRaise(re.compile('Statistics|STATISTICS')).touch()
    print("Pressed Stats tab at " + str(datetime.now().time()))
    vc.dump(window=-1)
    vc.sleep(_s)

def backFromDevice():
    vc.dump(window=-1)
    device.press('KEYCODE_BACK')
    print("Pressed Back at " + str(datetime.now().time()))
    vc.dump(window=-1)
    vc.sleep(_s)

if __name__ == "__main__":
    serialno = '.*'
    kwargs1 = {'ignoreversioncheck': False, 'verbose': False, 'ignoresecuredevice':
False, 'serialno': serialno}
    device, serialno = ViewClient.connectToDeviceOrExit(**kwargs1)
    kwargs2 = {'forceviewserveruse': False, 'useuiautomatorhelper': False, 'ignoreui
automatorkilled': True, 'autodump': False, 'startviewserver': True, 'compresseddump'
: True}
    vc = ViewClient(device, serialno, **kwargs2)
    main(sys.argv[1:])
```

D.2 iOS

```
function expandHomeGraphsNative(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  app.tabBar().buttons()["Stats"].tap();
  window.tableViews()[0].cells()[0].tap();
  window.tableViews()[0].cells()[1].tap();
  target.delay(5);
}

function expandHomeGraphsRN(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  target.frontMostApp().tabBar().buttons()["Statistics"].tap();
  target.delay(1);
  window.scrollViews()[0].elements()[2].tap();
  window.scrollViews()[0].elements()[1].tap();
  target.delay(5);
}

function expandRadiatorGraphsNative(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  var list = window.tableViews()[0];
  list.cells()[0].tap();
  target.delay(1);
  list.cells()[2].tap();
  app.tabBar().buttons()["Stats"].tap();
  window.tableViews()[0].cells()[0].tap();
  window.tableViews()[0].cells()[1].tap();
  target.delay(5);
}

function expandRadiatorGraphsRN(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  window.scrollViews()[0].elements()[1].tap();
  target.delay(1);
  window.scrollViews()[0].elements()[3].tap();
  target.delay(1);
  app.tabBar().buttons()["Statistics"].tap();
  target.delay(1);
  window.scrollViews()[0].elements()[2].tap();
  window.scrollViews()[0].elements()[1].tap();
  target.delay(5);
}

function turnOnOffDeviceNative(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
```

```
  var list = window.tableViews()[0];
  list.cells()[0].tap();
  target.delay(1);
  list.cells()[0].tap();
  target.delay(1);
  window.switches()[0].tap();
  target.delay(1);
  app.navigationBar().leftButton().tap();
  target.delay(1);
  window.logElementTree();
  list.cells()[0].switches()[0].tap();
  target.delay(1);
}

function turnOnOffDeviceRN(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  window.scrollViews()[0].elements()[1].tap();
  target.delay(1);
  window.scrollViews()[0].elements()[1].tap();
  target.delay(1);
  window.switches()[0].tap();
  target.delay(1);
  window.elements()[4].tap();
  target.delay(1);
  window.scrollViews()[0].elements()[1].tapWithOptions({tapOffset:
{x:0.76, y:0.35}});
  target.delay(1);
}

function selectRoomNative(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  var list = window.tableViews()[0];
  list.cells()[0].tap();
}

function selectRoomRN(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  window.scrollViews()[0].elements()[1].tap();
  target.delay(1);
}

function selectDeviceNative(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  var list = window.tableViews()[0];
  list.cells()[2].tap();
  target.delay(1);
```

```
}

function selectDeviceRN(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  window.scrollViews()[0].elements()[3].tap();
  target.delay(1);
}

function selectStatsNative(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  app.tabBar().buttons()["Stats"].tap();
  target.delay(1);
}

function selectStatsRN(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  target.frontMostApp().tabBar().buttons()["Statistics"].tap();
  target.delay(1);
}

function backNative(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  app.navigationBar().leftButton().tap();
  target.delay(1);
}

function backRN(){
  var target = UIATarget.localTarget();
  var app = target.frontMostApp();
  var window = app.mainWindow();
  window.elements()[4].tap();
  target.delay(1);
}
```

Appendix E

Questionnaire

UEQ

Please make your evaluation now.

For the assessment of the product, please fill out the following questionnaire. The questionnaire consists of pairs of contrasting attributes that may apply to the product. The circles between the attributes represent gradations between the opposites. You can express your agreement with the attributes by ticking the circle that most closely reflects your impression.

Example:

attractive	○	⊗	○	○	○	○	○	unattractive

This response would mean that you rate the application as more attractive than unattractive.

Please decide spontaneously. Don't think too long about your decision to make sure that you convey your original impression.

Sometimes you may not be completely sure about your agreement with a particular attribute or you may find that the attribute does not apply completely to the particular product. Nevertheless, please tick a circle in every line.

It is your personal opinion that counts. Please remember: there is no wrong or right answer!

	1	2	3	4	5	6	7	
annoying	○	○	○	○	○	○	○	enjoyable
	1	2	3	4	5	6	7	
not understandable	○	○	○	○	○	○	○	understandable
	1	2	3	4	5	6	7	

creative	○	○	○	○	○	○	○	dull

	1	2	3	4	5	6	7	
easy to learn	○	○	○	○	○	○	○	difficult to learn

	1	2	3	4	5	6	7	
valuable	○	○	○	○	○	○	○	inferior

	1	2	3	4	5	6	7	
boring	○	○	○	○	○	○	○	exciting

	1	2	3	4	5	6	7	
not interesting	○	○	○	○	○	○	○	interesting

	1	2	3	4	5	6	7	
unpredictable	○	○	○	○	○	○	○	predictable

	1	2	3	4	5	6	7	
fast	○	○	○	○	○	○	○	slow

	1	2	3	4	5	6	7	
inventive	○	○	○	○	○	○	○	conventional

	1	2	3	4	5	6	7	
obstructive	○	○	○	○	○	○	○	supportive

	1	2	3	4	5	6	7	
good	○	○	○	○	○	○	○	bad

	1	2	3	4	5	6	7	
complicated	○	○	○	○	○	○	○	easy

	1	2	3	4	5	6	7	
unlikable	○	○	○	○	○	○	○	pleasing

	1	2	3	4	5	6	7	
usual	○	○	○	○	○	○	○	leading edge

	1	2	3	4	5	6	7	
unpleasant	○	○	○	○	○	○	○	pleasant

	1	2	3	4	5	6	7	
secure	○	○	○	○	○	○	○	not secure

1	2	3	4	5	6	7

motivating	○	○	○	○	○	○	○	demotivating
	1	2	3	4	5	6	7	
meets expectations	○	○	○	○	○	○	○	does not meet expectations
	1	2	3	4	5	6	7	
inefficient	○	○	○	○	○	○	○	efficient
	1	2	3	4	5	6	7	
clear	○	○	○	○	○	○	○	confusing
	1	2	3	4	5	6	7	
impractical	○	○	○	○	○	○	○	practical
	1	2	3	4	5	6	7	
organized	○	○	○	○	○	○	○	cluttered
	1	2	3	4	5	6	7	
attractive	○	○	○	○	○	○	○	unattractive
	1	2	3	4	5	6	7	
friendly	○	○	○	○	○	○	○	unfriendly

	1	2	3	4	5	6	7	
conservative	○	○	○	○	○	○	○	innovative

På svenska

In English