Evaluating performance of a React Native feature set

Jesper Söderberg
Erik Johansson

Tutor: Dennis Persson
Examinator: Jody Foo
Upphovsrätt


Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility. According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Jesper Söderberg
© Erik Johansson
Evaluating performance of a React Native feature set

Jesper Söderberg
Linköping, Sweden
jesso429@student.liu.se

Erik Johansson
Linköping, Sweden
erijo422@student.liu.se

ABSTRACT
React Native has quickly become one of the most popular cross-platform frameworks for developing Android and iOS applications since it was first released by Facebook in 2015. In this study a proof of concept application is developed and a set of features are evaluated with regard to performance efficiency. Results of measurements show that while React Native does perform worse than its Android counterpart, it performs sufficiently well for building most applications with but can be more problematic for applications running heavier client-side logic.

Author Keywords
React Native; cross-platform; performance; frames per second; proof of concept; limitations; native module;

INTRODUCTION
The mobile phone industry has become a vast industry and its user base has become split between different devices that utilize different operating systems. For a software development company to reach this divided customer base they would have to develop separate applications for the different operating systems, which could become expensive to develop and maintain.

A solution to this problem is to use a mobile cross-platform framework which allows you to develop an application that works on multiple operating systems. The purpose of this paper is to construct a proof of concept application from a feature set inspired by another application and evaluate if one of the most modern of these kinds of frameworks, React Native is a suitable framework for the implementation of such an application. The chosen evaluation method is to analyze performance in comparison to native Android.

React Native
React Native is being developed by Facebook and was first released in 2015. React Native works for both Android and iOS devices, and allows you to develop an application in React that is then compiled into native applications for Android or iOS. This allows you to develop and maintain one application that works with both Android and iOS, instead of having to develop and maintain two separate applications for the different operating systems.

There are however a few potential downsides to consider before switching to React Native. The framework is another layer that could potentially have bugs and technical limitations, requiring you to wait for Facebook to fix it or find a way to circumvent the bugs. Another problem might stem from the fact that Facebook released React Native in 2015 which makes it relatively new, this in turn has given it less time to develop and might therefore be more difficult to find good documentation for certain parts. Some applications will need features written in native Android or iOS but in most cases everything should be able to be implemented without having to write native code.

Research Question
In our article we will determine important features in the native applications that we will attempt to port over to React Native from an already existing application that is owned by the company TaxiCaller. We will then measure the performance of the implementation of the features and compare them with the original application, in an attempt to get a better understanding of how suitable React Native is as a framework when trying to implement the set of features that we have hand-picked from the original application.

Suitability of using React Native when implementing a feature set with regards to performance efficiency of the implementation.

The suitability is based on appropriate performance efficiency metrics, which are chosen on a per feature basis. In the discussion we will look at further potential problems that might occur when using React Native that might need to be considered before deciding if it is the right framework to use for building a certain application. Performance efficiency was selected because it is important to considered the performance of a framework before porting or developing an application for a framework. Our study aims to give developers a helping hand in deciding if they should use React Native.

The Feature set that we have been selected based on the application we were given to make a corresponding React Native version of and are all important features of that application. Some features both in the original version and the React Native version might have been altered to be easier to measure during our tests. The feature set we chose is the following:

Scrollable List
Having a list is in a lot of applications a very common feature when you want to present things in a scroll-able view. Due to how common this kind of UI component is in applications it is important to see if there are any potential problems with having a list in React Native. The current TaxiCaller application uses lists in several places and it is of importance to have lists that perform efficiently.
Screen Transition
Transitioning between different screens is an essential part of almost any application. Therefore a poorly performing navigation system will affect the entire feel of an application in a very negative way. This feature involves screen changes (triggered by buttons or other actions) and a back action (triggered by a hardware button or a button in the upper left corner).

Background Process
Being able to run tasks in the background could be a requirement for certain types of applications. In TaxiCaller’s application there is a need to be able to ping their server on a frequent basis to keep the session active with the current back-end implementation. This feature is therefore considered quite important for the application to work as intended which is one of the reason it was chosen. For us pinging the server on a frequent basis equates to approximately every 5 minutes. Another requirement for this feature to be fully implemented is the possibility for it to work when the application is in the background. It is also likely that more background processes would need to be added if the whole of the native application would be ported over to React Native in the future. So there might be more complex requirements that could be part of certain applications when it comes to background processes that we won’t cover in this article.

Native Module
Last of the features that were chosen is a native module. This was chosen due to the fact that not all features are supported by React Native but should be able to be implemented with native code instead and then connected to JavaScript. In our project we decided to focus on a Java module instead of an Objective C module for iOS. The reason we chose to do this was due to the time constraints that we had and between Android development and iOS development we have more experience working with Android so it should therefore be easier for us to implement. While the original application included a module that was meant to listen to user-datagram-packages we decided to switch this into a module that runs Fibonacci computations to get more noticeable values during the performance test.

Background
Historically developing applications for smart-phones has required knowledge in different programming languages and development environments depending on the operating system used, iOS (Objective-C or Swift) and Android (Java). One common way to circumvent this knowledge requirement for different operating systems is to develop mobile web applications. However these applications tend to perform worse than native applications\(^1\) and for more advanced features you would still have to partially write native code therefore making a hybrid app.

Another way to deal with this problem of the developer requiring knowledge of several systems is to instead make cross platform applications. However historically there hasn’t been a good track record of frameworks that have managed to do this and perform well enough to compete with native applications.

Facebook’s React Native is seen as having much potential as far as cross platform frameworks are concerned. Being developed by Facebook and released two years after they released React, a JavaScript library for building user interfaces. React Native is already being used in a multitude of popular applications for example Facebook, Instagram, Uber, Discord and Skype.\(^2\) Though studies show that React Native in terms of performance doesn’t perform as well as native applications (see section “related work”), but it doesn’t fall much behind in performance either and manages to uphold a native feeling.

We will be working with TaxiCaller for this study. TaxiCaller is an IT company that specializes in cloud based traffic control systems for taxi- and transportation systems. They currently have a presence in 60 countries spanning 6 different continents.\(^3\)

Part of their current system include two mobile applications, each with a version for Android and another for iOS. Maintaining and developing the four versions of these applications takes a lot of time and effort which in turn makes them costly. It should for that reason be beneficial for TaxiCaller to adopt React Native instead. This would allow them to develop one application for React Native, which with some extra work for the specific platforms can be compiled to both iOS and Android applications. An important part of our work is that we create an application that will constitute a good proof of concept for porting the current application over to React Native.

The feature set that we will be focusing on implementing and evaluating is derived from the native application that TaxiCaller wants us to make a proof of concept for before converting to React Native. We have chosen a handful of features which are varied in nature. The choice to focus on these features arose from studying the application we had been asked to port and pick out features that had to be implemented at one point or another to give the same functionality as the TaxiCaller application. Sometimes we have altered the features of the original application to better be able to test them. Our intention was to look at features that could make for a bad experience for either developers or users if they do not perform relatively close to their native counterparts.

THEORY
There have been four different approaches to the development of cross-platform applications according to CP Rahul and S. Tolety[11]. The four approaches are web, hybrid, interpreted and cross compiled. While they all aim to overcome the same problem of having multiple code bases for different platforms they have different advantages and challenges.

Web approaches are web applications executed in the browser of the application and is therefore limited to the browser environment not being able to utilize platform specific features.

\(^1\)\[https://www.lifewire.com/p-2373173 (last accessed 2018-05-13)\]


Hybrid approaches are part web application and part native code for platform specific features. But usually suffer in performance since the executions happen in the browser engine.

Interpreted approaches execute the code in runtime on the platform accessing the native features via an abstraction layer.

Cross compiling approach allows you to compile your code for different platforms but makes you unable to reuse code for platform specific features.

React Native is a newer approach to cross-platform frameworks from those described in [11] but it resembles the interpreted approach the most however it uses API calls for rendering the application which is not explained in the interpreted approach. React Native runs JavaScript on top of the native iOS or Android OS. Since JavaScript is not the native language of either of these two platforms you need a way to execute the code. The technique to do this is commonly known as bridging and is not unlike how normal React interacts with the virtual DOM. By running the "JavaScript Core", which is a runtime environment for JavaScript that when run on top of the native platform allows us to be able to run JavaScript code on the platform. See Figure 1 for a visual explanation of the architecture.  

---

**React Native packages**

**Redux**

The Redux library for React Native is a JavaScript library that implements the flux architecture that was created by Facebook. One of the purposes of flux is to allow you to easier manage the data flow by allowing the data only to be sent one way. In Redux the state is stored in what is simply known as the store. You change the state by dispatching actions which are then caught by the Reducers which in turn modify the state.  

**Redux-Logic**

Redux-Logic is a library that complements Redux by handling logic in an asynchronous manner. It allows you to create Action side-effects, fetch data and intercept actions. This lets applications dispatch actions from a component to both change the state with a reducer as well as perform business logic, like API or database calls.

**List components**

ListView is the early version for handling lists in React Native, however this version is now deprecated and has been replaced by FlatList and SectionList. FlatList is the most basic of lists of the two. FlatList only renders items that are on the screen at the moment, and doesn’t keep the items that are outside of the view within the memory. This allows for more efficient utilization of the memory. However the downsides to this is that you instead need to store the items in the list outside of the list for example within the Redux state. Pure components unlike normal components are more optimized for re-rendering since they automatically have an implementation of a method called shouldComponentUpdate(). This method does a shallow compare between the new state and the previous state to see what needs to be re-rendered, and allows it to avoid re-rendering identical information. However be aware that shallow copies do not always trigger a re-render on certain elements in the list.

**React-Navigation**

React-Navigation is a React Native community solution for screen navigation within React Native and is even recommended by Facebook in the official documentation. It allows you to navigate between different screens on both iOS and Android versions of the application unlike NavigatorIOS. React-Navigation allows for easy to override navigation logic that integrates well with Redux. It also uses the React Native animated library to allow for animations that can be customized and allows for easy applicability of custom headers to pages. A few of the different types of navigator components are for example stacks, tabs and drawers.

---

**Performance Efficiency**

Performance efficiency is according to the ISO standard defined as "performance relative to the amount of resources used under stated conditions".[1] Different sub category qualities are relevant depending on the feature being observed, it is therefore important that we look at each feature and try to determine which metrics of performance efficiency are the

---

4. [https://www.logicroom.co/react-native-architecture-explained](https://www.logicroom.co/react-native-architecture-explained) (last accessed 2018-05-13)


7. [https://github.com/jeffbski/redux-logic/blob/29ffe28b46ae1fe9dcd7f130bff43a206536ba7/README.md](https://github.com/jeffbski/redux-logic/blob/29ffe28b46ae1fe9dcd7f130bff43a206536ba7/README.md) (last accessed 2018-05-13)

most relevant to determining the performance of the feature in the application. Performance efficiency plays an important role in deciding which framework to use since we do not want applications with poor performance.

Related work

Cross-platform

An article by Ahamed, S. I., Pezewski, A., and Pezewski, A.[2] where they attempt to explore the subject area and select different criteria which might be of interest to look at when selecting a framework. Some of the criteria they decided might be useful were the domain of the framework, the ability to extend the framework, coupling, language requirements and platform requirements. In our article we have touched upon which the intended domain for React Native is, the language requirements, platform requirements and also the ability to extend the framework using native modules.

D. Garlan, R. Allen and J. Ockerbloom [5] look at architectural mismatches when a developing a system from different parts. The article demonstrates the problems of developing these kinds of systems by constructing a prototype using a variety of different parts all written in c++ or C and with extensive usage by companies. Their results show that these systems can cause a lot of mismatches and make it both time consuming to develop and create bad performance. They therefore recommend others to avoid using existing parts to construct a system when they can. In our work we will take a look at a Java module which can constitute as a different part then the rest of our system and can therefore cause mismatches on an architectural level.

React Native

A study by N. Hansson and T. Vidhall[6] evaluate and compare a React Native application to native Android and iOS applications. In their study they look at the performance efficiency and conduct a user experience study on the application. Evaluating performance efficiency by looking at CPU usage, memory usage, response time, frames per second and application size. Response time was measured by conducting user tasks and measuring the response time of the application during these tasks, while frames per second, CPU usage and memory usage were tested via performance test scenarios. There were three performance test scenarios that were ran using automated test scenarios to make them easier to replicate. Their comparison shows that React Native performs worse than the native counterparts but that the difference in performance was smaller than expected.

A similar study is done by F. Carlström and O. Axelsson[3] in which they attempt to evaluate a React Native application in a large extent of different ways including measuring performance efficiency, conducting a user experience study, testing the reliability and by observations during the development process. In terms of performance efficiency in their case they look at CPU usage, memory usage, battery, network and application size. It is clear that the benchmark performance test was not the main focus of the work in this study and it is therefore just a simple benchmark test on the final product where they measure the usage of different resources in the application. They conclude that while React Native performed worse than the native applications the historical gap that has been between cross-platform frameworks and native code has shrunk a lot and it is therefore a viable option to use React Native. In this article they also go through their implementation process in detail, where the time taken to implement a specific part of the application in both React Native, native iOS and native Android has even been given.

Finally in an article by A. Lelli and V. Bostrand[9] we notice a slightly different method. While the study itself focuses on performance efficiency the article looks at different scenarios. These scenarios are extremely similar to what we call features. These features vary from graphically intense activities, application navigation and a ListView. They are evaluated with different metrics depending on the feature and are all compared to an Android native counterpart. Since the ListView is now a deprecated technology in React Native and has been replaced by FlatList which we will evaluate in our study, it will be interesting to see the performance difference between the two. The Method that is used when evaluating the ListView looks at two different ListView implementations. One containing a list of contacts including image, name and phone and styled to look like a card and the other implementation represents a photogrid with each element being 3 images. They are tested having 500, 1000 and 1500 elements and look at the frames per second when scrolling. The results of this feature shows that React Native performs worse than native applications in terms of memory usage, CPU usage and frame rate stability it does however take less GPU usage. Navigation, startup time and battery is evaluated creating a “budget divider application” which is based on an application called “Tricky Tripper”. It is believed that this will be more of a realistic model for evaluating these metrics since you got several components being used in one application. Overall the article concludes that the performance of React Native is worse than native counterparts, but is still a viable option for applications that do not rely on heavy calculations.

Evaluation

C. Ryan and P. Rossi[4] assess context-aware mobile application performance and resource utilization. By specifying different software attributes and defining a concrete metric and measurement technique to each attribute and testing a series of proposed hypothesis using linear correlation analysis. Results show the impact of the attributes on the efficiency of context-aware mobile applications. [4]proposes a formal model of mathematical equations in order to facilitate dynamic runtime placement of mobile objects and tests the model on a practical application. Concluding that the local adaptation of the strategy results in higher efficiency related to performance and resource utilization.

J.K. Lee and J.Y. Lee [8] propose guidelines on how to program Android applications more efficiently. By comparing the performance between two different programming languages used for creating Android applications namely Java and Native C. Looking at how the languages perform on five different categories; JNI delay, Integer, Floating-point, Memory access algorithm and String processing. The results show
that Native C was faster on all categories except for String processing.

C. Ming, J. Lin, C. Dow and C. Wen[10] benchmark Dalvik Java code and native code for Android. Conducted twelve different tests to analyze the performance, concluding that native code is faster for about 34.2% while Dalvik Java code was faster in three of the tests and in one test according to [10] the Dalvik Java code was said to perform very bad.

In a study by C. Hsieh, H. Falaki and N. Ramanathan[7] evaluating the performance of Android Inter-Process Communication (ICP) for Continuous sensing applications by constructing a benchmark application and measuring the performance of a)Latency, b)Memory footprint and c)CPU usage.

METHOD

Implementation

We will implement a proof of concept mobile application in React Native that has all of our features. The following have been provided to us by TaxiCaller and have been utilized during our development:

- Current TaxiCaller Android/iOS code
- Code skeleton and common features of another application in development at TaxiCaller
- TaxiCaller back-end API

All development was done at TaxiCaller’s office where we could get help with their systems. All features except for the Java module were developed to be functional for both iOS and Android.

The provided code skeleton includes router navigation, persistent storage, network actions, styled components and more common smaller components. This skeleton was provided to us by TaxiCaller and allowed us to spend more time on implementing our feature set. TaxiCaller also provides us with a back-end which the application communicates with for all network actions.

React Navigation

In our implementation of the navigation system we choose to go with React Navigation due to it being a popular option and was not run using native code. By dispatching navigation actions we are able to navigate between different screens. We also store the last screen that we navigated to, to prevent re-navigation to the same screen without having navigated to another screen in between.

The code Snippet in Figure 2 shows how navigation is used inside the Redux reducer.

FlatList

The code snippet in Figure 3 shows how the FlatList is added into a view. InitialNumToRender defines how many items are initially rendered. We chose eight items because that is how many items filled just over the height of the screen. Keeping this at a low value reduces the render time when the list is first shown. getItemLayout is an optimization that specifies the height of each element manually which results in smooth scrolling with no loading. Every item in the list will be of the ShiftCard type which is a pureComponent. Data and button press handling is passed along to the ShiftCard component.

React Native Background process

Our implementation of the background process was done by utilizing Redux Logic. As can be seen in the code in Figure 4 by using Redux Logic we created a process that is activated when it receives a specific action and cancelled when it receives another specific type of action. In this process we then use a setInterval method to call the API once every interval while the code has been activated.

```javascript
const INTERVAL = 60000;
export const bgProcessLogic = createLogic({
  type: NetworkActionTypes.REQUEST_SUCCESS,
  cancelType: localActionsType.SIGN_OUT,
  warnTimeout: 0,
  validate({getState,action}, allow, reject){
    action.requestType === "SIGN_IN" ?
      allow(action) : reject(action);
  },
  process({ cancelled$ }, dispatch, done) {
    const tick = setInterval(() => {
      dispatch(
        apiActions.sendProcess({})
      );
    }, INTERVAL);
    cancelled$.subscribe(() => {
      clearInterval(tick);
    });
  }
});
```
**Java module**

Our current Java module implementation is a process that is computing Fibonacci. At first we were going to listen for User Datagram Protocol packages on a specific port, after receiving a package we would process the package so that we could retrieve the appropriate content via an API call. But instead we decided to use Fibonacci numbers computations due to the fact that it left a much larger footprint then our current implementation on the CPU usage which made it easier to compare the results and Fibonacci is easy to implement in almost any language therefore making it easier to compare with other frameworks in the future. The Fibonacci computation is recursive and calculates the first 30 numbers.

To get access to our Java module in React Native we exported some of the functions to React Native functions which are then called to interact with the Java module.

**Environments and tools**

All our tests will be performed on an Android system with both our React Native application and the native Android counterpart. We will repeat the tests five times each to see if there are any deviations in performance and to give a more reliable result. The physical device that will be used is a OnePlus Two smartphone running Android version 6.0.1 / OxygenOS version 3.6.1. The OnePlus Two features an octa-core snapdragon 810 processor, 4 GB RAM and an Adreno 430 GPU. The most relevant node module versions are react-native v0.54.2, react-navigation v1.5.8 and redux logic v0.12.3.

We will be using GameBench as the tool for measuring and analyzing the performance. This tool allows us to look at a multitude of different metrics during our tests. GameBench runs on Windows and installs a service on the Android device which it then communicates with over USB. This service does not affect the application being tested and every metric we will be using is for the application and not the devices’ total resource usage. On Android, GameBench uses data provided by the standard proc filesystem which is used by other utilities for real-time monitoring. Because the CPU that will be used is an octa-core, 12.5% (100% / 8 cores) total usage means that one core could be running at 100% and cause issues.

**Evaluation method**

In this section we will talk about what qualities should be evaluated, why we chose those qualities and how they will be measured. To evaluate our features’ performance efficiency, a method similar to that of T. Vidhall[6] is the most appropriate. This method will be based on running test scenarios were we measure the different qualities. We decided against using automated test scenarios due to technical difficulties and time constraints.

**FlatList**

When picking the qualities for the FlatList evaluation we tried to include the qualities used in the study by Lelli that were used to evaluate the ListView. Since FlatList is the replacement for the now deprecated ListView it will be valuable to be able to compare performance. The qualities that we look at are CPU usage, GPU usage, memory usage and frames per second. Since the FlatList loads and renders elements when scrolling the list these are the most important qualities to measure.

Our test scenario is based on the study by A. Lelli[3]. In this test scenario we will have 500 list elements containing some formatted text labels. The test will be started with a two second idle period before we start manually scrolling through the list for 15 seconds, followed by another two second idle period.

**Screen Transition**

The most relevant qualities to look at when it comes to navigation are those that indicate how well the navigation handles transitioning between activities. We chose the metrics CPU usage and GPU usage because similarly to FlatList, during transition a new activity will be loaded and rendered in the application.

This test scenario is based on the transition phase between two screens, since this is the main purpose of navigation. The test will be started with a two second idle period followed by a button press to navigate, a three second delay, a back button press and a two second delay.

**Background Process**

In terms of performance efficiency we do not expect any extreme performance differences in our current implementation of a background process. Since our background process only pings the server it is a very light task and should barely affect performance at all. If however we had a background process that was a heavy computational process we could potentially see a big performance hit on the CPU. We will therefore look at CPU usage since it is used during background computation unlike the GPU.

The test scenario for this feature is simple, it will run for 30 seconds and only consists of a ping call to the server, which will be called every 500 milliseconds. We chose a low frequency of 500 milliseconds to simulate more network traffic running in the background. We will also measure the application in an idle state for the same amount of time. Then we should be able to compare the results and see the spikes in performance between when the background process is being run and the application is idle.

**Java module**

Since the code is the same code in both React Native and native Android we do not expect to see a noticeable difference in it affecting performance. However since this module does no rendering and mainly works similar to a background process we will only look at the CPU usage.

We should expect a very similar result when the code is being run independently to the native Java application. Our test scenario will consist of running Fibonacci calculations frequently to simulate the heavy calculations some applications
need to run. This math will run approximately every 100ms for the 30 second test.

RESULTS

FlatList
When we look at the results of comparing the React Native Flatlist with Android list in Figure 5 we can quite clearly see that the CPU usage on the FlatList implementation is substantially higher than the Android version. However the Android list does have a higher utilization of the GPU which is most likely not a bad thing as we will discuss later.

The median frame rate of the Android version was 57.8 and for FlatList it was 51. Both applications ran at a stable frame rate and there were no visible stuttering or drops in framerate that could be felt when scrolling.

The memory usage of the applications can be seen in Figure 6 and yet again show Android outperforming React Native by using less of the memory both during the application being idle and while scrolling the list. The difference in the memory increase between idling and scrolling is far less the Android version, being only 20% compared to the difference that React Native displays which is 35%.

![Figure 5. CPU & GPU usage when scrolling the list](image)

<table>
<thead>
<tr>
<th></th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU STDEV</td>
<td>0.156</td>
<td>0.166</td>
<td>0.19</td>
<td>0.158</td>
<td>0.164</td>
</tr>
<tr>
<td>CPU AVG</td>
<td>0.345</td>
<td>0.331</td>
<td>0.341</td>
<td>0.341</td>
<td>0.339</td>
</tr>
<tr>
<td>Android Idle</td>
<td>0.163</td>
<td>0.151</td>
<td>0.123</td>
<td>0.162</td>
<td>0.136</td>
</tr>
<tr>
<td>CPU STDEV</td>
<td>0.175</td>
<td>0.151</td>
<td>0.144</td>
<td>0.155</td>
<td>0.142</td>
</tr>
<tr>
<td>Android CPU AVG</td>
<td>0.233</td>
<td>0.208</td>
<td>0.243</td>
<td>0.246</td>
<td>0.242</td>
</tr>
</tbody>
</table>

Table 1. Background and idle CPU average and CPU standard deviation from the test runs

Background Process
The background process results show us that the CPU usage is higher in React Native both when the application is in an idle state as well as when pinging the server frequently which can be seen in Figure 7.

By looking at the standard deviation in Table 1 of the test results we can see that the deviation is much higher for React Native when running the background task than Android. In fact the idle deviation and background process deviation is about the same for Android. Which should imply that Android does not get as huge peaks as React Native in CPU difference. The growth in CPU utilization between idle and pinging is about 2.68 times. Compared to the growth of the Android version which is much lower being only 1.56 times between idle and pinging.

![Figure 7. CPU usage when idle and when pinging at 500ms delay](image)

Screen Transition
Results from the navigation, just like previous results, show that React Native has a higher CPU usage than the Android version as we see in Figure 8. And just like the FlatList the Android version instead has a higher utilization of the GPU.
There was a drastic difference in terms of CPU usage when it comes to the Java module as can be seen in Figure 9. React Native CPU usage averaged 5.7% while Android was at a much lower 0.49%.

The frame-rate is a bit lower on the React Native application but due to the problems when measuring FPS (see discussion method) the results might be less accurate than expected and since the difference is rather small we did not notice a difference ourselves.

The memory usage is also a bit worse on the React Native application and the growth rate of the memory usage is also higher which tells us that the Android view for lists is more efficient or potentially that React Native as a whole does not deal as well with memory headers. Since Android Java code allows you greater responsibility over the memory usage this probably allows for greater optimization in terms of memory compared to React Native which is written in JavaScript.

What is not shown in the measurements is that React Native has difficulties keeping up with list item rendering when scrolling fast. When the list is used normally and is quite short this is not an issue. But with long lists and the user wanting to scroll up or down to skip past most of the content in the list, the rendering will be too slow. While the user is not actually reading the content when scrolling this fast it it takes a while to render when stopping which might cause for a worse user experience.

Screen Transition

Just like previous results we can once again see that React Native does not utilize the GPU as much as the Android application.

When developing using React Navigation we ran into one issue. When quickly tapping a navigation button the application would dispatch several actions and create duplicate screens or go back several steps if the back button was pressed quickly. On Android the hardware back button also had to be properly connected to the system. In the end we managed to fix this problem in a good and simple way that seems to have solved the "bug". Our Redux state stores the screen last navigated to and it can not be navigated to again without another navigation action in between. The back button can now only dispatch an action once for every time a screen mounts which solves the double tap back clicking that caused the application to skip back to the sign in menu.

Due to large amount of different packages for managing navigation it means that there are potentially a lot of optimization that can be done and packages that present more performance efficient solutions. Most likely the React Native Navigation package will perform better then our current implementation since it utilizes more native code which can be seen in our result is usually better in terms of performance.

We also believe from our experience and due to the large amount of different packages that more advanced navigation features should not cause a huge problem for creators due to the flexibility that React Native has in terms of navigation.

Transition animations used by React Navigation we did not optimize appropriately instead we opted to disable them to speed up the navigation and to factor them out of the tests. This gave the navigation of both applications a similar feel even though the Android version has a slight fade in animation when navigating. It is possible that these animations could be tweaked to work better but we prioritized performance and minimal transition delay.
of CPU usage. In fact the difference in CPU usage growth between idle and pinging the server is much larger in the React Native application. While our task is a very lightweight task resource wise if the CPU usage growth trend would be maintained for more resource heavy tasks it would make React Native a bad alternative in terms of resource heavy tasks. This is also shown in the article [9] where they conclude that React Native is not fit for handling resource heavy tasks.

In terms of solutions for implementing the background process we had two other options that we decided against. One was using a setInterval() inside of a component but this felt very incorrect due to the fact that we want to separate the logic side of things and the components. The other option was to create a worker that would work on a different thread and call the API every interval. While this solution provides with more accuracy due to it being on a separate thread it does make it harder to dispatch actions since the worker state is separate from the main application state.

A very important problem arises when you have slightly different requirements for this feature. If the feature has a requirement that the background process should be run every 5 minutes while the application is in the background state of the phone it will not be able to do this straight out of the box. All packages that are meant for background process only allow background processes in the background state to run every 15 minutes. This is due to a restriction on iOS devices where normally application are only allowed to run things in the background every 15 minutes unless they are registered as needing a specific type of background process like playing music or using the GPS. The packages for React Native however are either built for the strictest requirement and only allow for background process to be run every 15 minutes or are solutions exclusive to the Android version. It is unclear if there is a viable solution for this on iOS devices especially if you still have the desire for the process to be run inside the React Native code and not a iOS module. More research is required to determine whether or not it is possible to do in React Native but we can state that it is currently not a simple solution that works by just installing a package but instead a more complex solution would be needed.

Java Module

The Java module results are to us quite unexpected, we had assumed that the Java module would perform almost identically to that of the Android application since they were both the same Java code. Instead the result showed a staggering difference in performance between the two, where the React Native Java module performs considerably less efficient than the Android version.

We are not at all certain as to why the React Native Java module performs so much worse. It might potentially be a problem in our implementation or it might be a difference in how React Native handles the Java modules.

Setting up a Java module that works with React Native is not that complicated. However in our case it took quite a lot of time and effort because we had to take already existing code from the Android application and try to decouple it and bring over the appropriate dependencies. We later decided to change the code to a Fibonacci calculation as to make a more noticeable print in the CPU usage.

Method

Properly measuring the mobile phone performance of an application proved to be a much more difficult feat than expected. We encountered problems both when trying to measure the GPU usage as well as the FPS. The GPU was difficult to test accurately because of the effect the interaction with the smart-phone screen had on the GPU usage. Every time we touched the screen it would cause a spike in GPU usage. These spikes would vary in how long they were depending on how long you touched the screen. So when we tried to do a test that involved us pressing a button the slight variation in time it took to press the button would change the result, causing the test to be less accurate.

Our other difficulty was measuring FPS because when nothing was rendering or moving on the screen the FPS would drop to 0 instead which causes the result to be worse then it actually should be. This could potentially be caused by performance optimization in the Android device but it does make benchmarking FPS more difficult.

We did not perform automated test scenarios due to problems with finding a working tool in the scope of time we had. Instead we did our test manually which most likely decreased the accuracy of our results. A good example of this is the GPU usage spikes, these were varied due to our human interaction with the screen if they were automated we would have had more consistent spikes and it could therefore be easier to ignore them in the results. The tests also varied in execution time between a few seconds because of the human element which might sometimes affect the results accuracy. This will also lower the replicability and the reliability of the results.

We are not certain how accurate the performance difference is since we only constructed one of the applications and the Android application that we compared with has a lot more features implemented that might affect the performance and have had several years in which it most likely has been optimized more than our React Native application.

Since large parts of the code that is used belongs to TaxiCaller it also makes it more difficult to replicate the exact tests however replicating the features should still be possible with your own implementation.

Due to time constraints we chose to implement a Android module only and not an iOS module. Given more time we would have wanted to implement an iOS module and also perform tests on an iOS device to give a more extensive performance analyzes.

The performance tests we ran were not very heavy on either CPU or GPU and it could have been more interesting to benchmark more resource heavy features.

Due to the problems explained above the results become less accurate however their validity is still fairly good. We have realized that it might not be optimal using averages when talk-
ing about things like CPU usage and GPU usage since they might give miss leading results but we have chosen not to include standard deviations for any more then one result. The reason being that the standard deviation is already expected to be very high for the other tasks since they are not constant in utilization of resources but include resting and active phases.

The academic sources that we have been using our article appear to be reliable sources of information. We largely chose the articles based on how they related to React Native apart from two of the articles which were chosen more due to them relating to the theme of framework suitability. We attempted to find articles that provided a fairly high replicability, reliability and validity and we succeeded fairly well in doing this. We have taken inspiration from certain articles both in the choice of what features to look at and in terms of evaluation method.

ETHICS

Our results could help people who are considering using React Native to determine whether React Native is a solid choice for building their application. Potential future work that could be of interest is creating possible solutions to solve background tasks when the application is in the background state on a more frequent basis then every 15 minutes. It would also be useful to see other studies taking on the problem of performance differences related to Java modules compared to Java to see if this is actually a big problem or maybe just an implementation related problem.

CONCLUSION

The aim of this thesis was to evaluate the performance aspects of React Native by creating a proof of concept application. The developed application was a port of TaxiCaller’s Android application and was compared to that application. Implemented features include scrollable list, screen transition system, background process and native Java module. Each feature was evaluated by measuring its performance and the implementation was discussed in regard to difficulties and optimization details.

The study concludes that React Native does perform worse in all tests but it is unclear if the difference in performance is large enough to affect most developers. The most important observation we can make is that there is a noticeable performance difference when comparing a Java module for React Native to a completely native Android app. There is also a large uncertainty whether it is possible to run frequent background tasks when the application is running in the background.

The benefits of using a cross-platform framework might still outweigh the weaker performance of React Native. Developing resource demanding applications in React Native is not something we can recommend since the performance difference may be more noticeable for those kinds of applications that utilize more resources.

REFERENCES

1. ISO/IEC 25010:2011 systems and software engineering - systems and software quality requirements and evaluation (square) - system and software quality models, 2011.