Bridging the Gap: Investigating Device-Feature Exposure in Cross-Platform Development

Andreas Biørn-Hansen
Faculty of Technology
Westerdals Oslo School of Arts, Communication and Technology
Oslo, Norway
bioand@westerdals.no

Gheorghita Ghinea
Department of Computer Science
Brunel University
Uxbridge, United Kingdom
george.ghinea@brunel.ac.uk

Abstract

By traversing academia and developer communities, two predominant approaches to cross-platform mobile development have been identified, specifically Hybrid and Interpreted. Previous research has established the use and integration of platform- and device-specific features to be core requirements for cross-platform frameworks. In this study we assess and discuss how the Hybrid and Interpreted approaches facilitate the use of native device features from within a JavaScript context, and how custom communication bridges are both developed and integrated. Our research motivation lies in data from an industry survey, stating that developers perceive device communication as a real pain-point. While both approaches exist to ease development of mobile apps, they are fundamentally different at a technical level. The article takes a technical approach, drawing evaluations and discussions from two app implementations. Our findings indicate that implementation and development of communication bridges are non-complex tasks, and that execution-time performance varies greatly.

1. Introduction

Within cross-platform mobile development, different approaches and associated technical frameworks exist to facilitate the development of apps that execute and work across platforms, often depending on just a single codebase. As such solutions are effectively layers of abstraction between the developer and the native part of an app, *bridges* are the enabler for communication between the native side and abstraction (app) side. The bridges and bridging system are integral parts of several development approaches, including the popular Hybrid and Interpreted ones. The bridging system facilitates the use of native device features including, but not limited to, Bluetooth, device storage and camera access [1]. These are features normally accessed through platform-specific code using the native development approach [2]. In native development, platform-specific languages such as Java or Kotlin for Android, and Objective-C or Swift for iOS are used to develop apps specific to a given platform [2]. If targeting multiple platforms, such as Android, iOS and Windows Phone, the increase in development cost is almost linear for each new platform added to the specification [3]. This is due to inherent and strong platform heterogeneity [4].

In this study, we present a technical comparison of how two popular cross-platform app development approaches, *Hybrid* and *Interpreted*, communicate with- and leverage device features. We base the comparison on two technical implementations, in our context being mobile apps. Specifically, the study investigate how the approaches, and associated frameworks, facilitate the creation and use of bridges to access native functionality from the *app-side* to the *native-side*. Thus, we extend and draw from studies such as [5]–[7] by deeper investigating how platform-specific code is executed via bridges from common JavaScript environments. Additionally, we present preliminary results from a conducted performance test, thus enabling us to draw from an empirical foundation for the sake of comparison and discussion.

Throughout this article, we refer to two sides of our app implementations. The *native-side* refers to platform-specific code, effectively the code that communicates directly with the given platform SDK and device features. This code is Java/Kotlin for Android, and Swift/Objective-C for iOS. The *app-side* refers to the codebase of our implementations a cross-platform developer would spend the majority of their time working with, in our context being JavaScript-based.

Based on our literature review in Section 2, we find that when discussing cross-platform mobile app development, we tend to sort technical frameworks into five particular development approaches. *Cross-platform* is thus used as an umbrella term rather than to describe a specific technology or way of development. Research seem to agree that included in this pool of approaches we find Hybrid, Interpreted, Cross-Compiled, Component-Based and Model-Driven development [1], [3], [5], [6], [8]–[11], where the two latter approaches are less frequently discussed in literature than the former three. While they all have a common goal

URI: http://hdl.handle.net/10125/50605
ISBN: 978-0-9981331-1-9

of easing mobile app development compared to traditional native development [11], how they do so differs on a technologically fundamental level.

The Cross-Compiled approach does not depend on a device communication bridge due to its nature, where a common language is compiled into platform-specific binaries [8], thus eliminating the need for bridges between languages. Thus, the Cross-Compiled approach was left out of this study. For the Component-Based and Model-Driven development approaches, we find academic contributions to be prevalent, with less industry efforts and adoption. Both Heitkötter *et al.* [3] and Perchat *et al.* [8] are examples of academic efforts increasing awareness of respectively Model-Driven and Component-Based app development. These approaches do not require device communication bridges either, and are thus left out of the study.

The Hybrid and Interpreted approaches both rely on bridges for communication between the app-side and native-side. Most Hybrid apps rely on Cordova, an open-source library for easy facilitation of bridge communication and app packaging. No such library has been identified for the Interpreted approach. An in-depth explanation of both approaches are presented in Section 4.

Our research question and motivation is based on an online survey questionnaire conducted on- and targeted towards industry developer communities. The survey included 101 respondents, and focused on different aspects of cross-platform development. When questioned regarding typical developer experience pain-points, 32 respondents identified and answered that "Hard to integrate with device APIs (camera, Bluetooth, file system, etc.)" was an issue they related to cross-platform development. Based on these findings, we formed the following

Research Question: *"How programmatically complex is the development and use of communication bridges allowing for access to platform-specific device features in Hybrid and Interpreted apps?"*.

As this article is more of a technical nature, the survey will not be presented in any further detail. However, presenting the survey is part of our planned future work, and will provide extensive insight into the status quo of developers perception of cross-platform tools, frameworks, issues and interest.

The rest of this paper is structured as follows. Section 2 presents a discussion of related work. In Section 3 we describe the research method used. We continue on to an in-depth description of the development approaches in Section 4. Section 5 describes the technical frameworks included, as well as a thorough technical description of the implementations. In Section 6 we present the findings from our preliminary performance study. We discuss our findings in Section 7, before concluding and suggesting future work in Section 8.

2. Related Work

An extensive body of research has been identified on the subject of cross-platform app development. The majority of well-cited papers discuss framework-level differences as a vehicle for comparing development approaches [1], [3], [5], [6], [8]–[10].

Technical artefacts and implementations are frequently included, substantiating the research and providing data where suitable [1], [5], [8], [12], [13]. In this study, we use technical artefacts to better understand and communicate the complexity of feature-access bridging on cross-platform development. However, implementations are used also for studies such as comprehensive performance tests (e.g. [14]) and comparison of development approaches (e.g. [13]).

In Puder *et al.* [15], the authors propose their own Cordova-like alternative for exposing device features to what they refer to as web apps, in our context being the Hybrid approach. While the proposal is seemingly novel and interesting by itself, the authors present Cordova in a rather opinionated fashion, stating that platform restrictions impose a negative impact towards the end-user without providing any additional proof of such. We are unable to find the proposed framework implemented in any production-released tool or framework. A similar alternative was presented in Bouras *et al.* [16], scrutinizing a technical implementation using their own Cordova-like framework. While being an interesting contribution, it lacks real-world adoption and testing, making it more a research effort than of direct relevance to industry and practice Nevertheless, both studies contribute greatly to the understanding of possibilities within the Hybrid approach, but less so for understanding real-world solutions.

We identified Heitkötter *et al.* [2] to be part of the fundamental research on cross-platform development. The study extensively scrutinize different perspectives of cross-platform development, including such as distribution platforms, Graphical User Interface designs, scalability and access to platform-specific features. The latter perspective, or criteria, is of great interest to us. Whereas the authors compare frameworks and approaches using their proposed set of criteria, we extend their findings by presenting implementations focusing on *access to platform-specific features*, using Cordova and React Native modules. The same criteria is also proposed by *Gaouar et al.* [6] in their article on cross-platform tools' desirable requirements. The attention feature access receives in the context of cross-platform development, together with the aforementioned survey results, further validates the demand for our proposed research question.

Furthermore, in an article by Majchrzak *et al.* [1], the authors investigate differences between three technical frameworks, including React Native, Fuse and Ionic Framework. Their technical implementations include numerous device features requiring bridges; proprietary Interpreted-bridges

for both React Native and Fuse, and Cordova for the Hybrid-based Ionic app. While they present certain evidence of the ease of integrating with device features, their focus is not on how such bridges work or how plugins/modules are developed, but rather on the technical artefact development itself.

From the industry, a book by Gok and Khanna [17] dives into the technical parts of the WebView component and its execution flow between Java and JavaScript. However, Cordova was merely mentioned as an alternative to their own Cordova-like implementation. The book provides an introduction to the general lower-level foundation of how WebView's operate, but nevertheless limiting the book's reach as Hybrid apps tend to use Cordova (Section 4).

The targeted gap was identified based on the presented literature review. We found a number of papers focusing solely on non-design comparisons of technical frameworks and development approaches. To the best of our knowledge, no identified research has focused in-depth on app-side-to-native-side communication in the context of cross-platform apps using established technologies and frameworks. Where related work from both industry and academia focused on implementing novel but adoption-limited approaches, we were unable to identify academic research focusing on more established technologies such as Cordova or similar industry-adopted alternatives in our context.

3. Research Method - Design Implementation

To provide a technical foundation for understanding app-side/native-side communication in the Hybrid and Interpreted approaches, a total of two apps were implemented using a framework of each approach. To facilitate the development, the Ionic Framework (Hybrid) and React Native (Interpreted) were used, both due to popularity and industry adoption. It is important to note that such frameworks may impose opinionated paradigms or design patterns, which may limit the validity of our findings to the set of used frameworks.

The Hybrid-based Ionic app is written in a JavaScript superset language, TypeScript. It builds on the new JavaScript standard, ECMAScript 2015, and adds types to the language. The Interpreted React Native app is written using the new JavaScript ECMAScript 2015 standard. The JavaScript code listings presented throughout the article will thus differ from traditional JavaScript.

4. Chosen Approaches

4.1. Hybrid

A fundamental principle of the Hybrid approach is to facilitate app development by exposing device features to a WebView component, and wrapping of the WebView

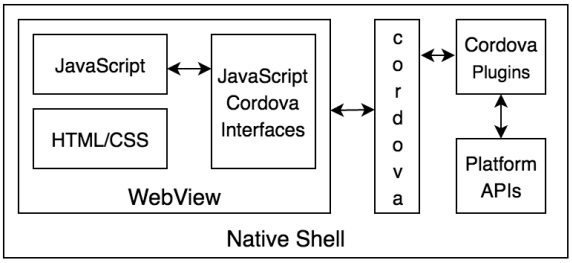

Figure 1. Hybrid approach architecture.

and app code into an app, also known as a Native Shell [15], [18]. Figure 1 illustrates the architecture of Cordova-based Hybrid apps. Due to such apps using regular web technologies including HTML, CSS and JavaScript, they depend on a WebView component for rendering content. This is another core feature of Cordova, as it by default implements a WebView component into the native shell. Because of the shell, Hybrid app can be downloaded from regular app stores. Cordova and the native shell cooperates to load the local HTML, CSS and JavaScript files, constituting the user interface and business logic of the app, into the embedded WebView component, which in turn acts as a render engine rendering content to the screen.

We have identified a number of technical frameworks belonging to the Hybrid development approach, such as PhoneGap, Ionic Framework and Onsen UI. As they all build on Cordova, they are not fundamentally different from each other. Their core focuses, however, differentiates them.

For an Hybrid app to communicate with device features such as camera, geolocation and device storage, it calls the feature's respective Cordova plugin, which the bridge handles. Certain plugins are included out-of-the-box as core plugins [19]. If a plugin does not exist as part of the core plugins package, community efforts are available and searchable at the Cordova Plugin page [20] with more than 2600 open-source plugins as of August 2017. If an application requires a plugin not found in either repositories, custom plugins can be developed to bridge the functionality needed. As discussed in section 5, a custom Cordova plugin is developed as part of our presented study.

4.2. Interpreted

Certain characteristics are fundamental to apps of the Interpreted approach. They differ from Hybrid apps in that the rendered user interface is actual native interface components [9], [21]. As such, the user interface is as native as a regular native app written in a platform-specific language would be. In comparison, the Hybrid approach renders HTML components to a WebView. Figure 2 illustrates the architecture of React Native. As discussed, the architecture and implementation of Interpreted frameworks differ, thus

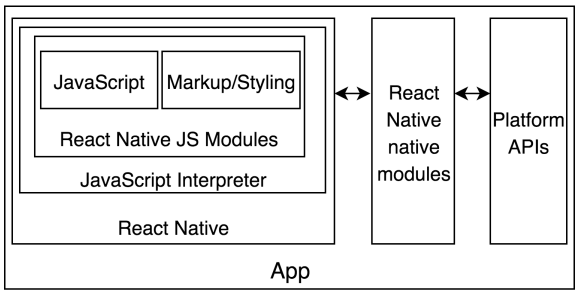

Figure 2. React Native/Interpreted approach architecture.

the illustration may not be valid for other frameworks of the same approach. Nevertheless, the figure illustrate the parts of a widely popular Interpreted framework.

The rendering of native user interface components is made possible due to the user interface bridge dispatching layout calculations to the native-side of the app [10]. An Interpreted app consists mainly of two parts; one that interfaces the app developer and provides tools required for app creation and development, and one that handles the native-side and interpretation of code from the former part. Code interpretation is true for both user interfaces and device-feature access, such as Bluetooth, camera and device storage. If an Interpreted app require access to a native feature such as the device's camera, it will pass the request down through the respective bridge handling such requests, and return a result up through the same bridge where needed, e.g. to return a value from the native-side to the app-side.

Upon starting an Interpreted app on a device, the app-side codebase is Interpreted (hence the approach name) by the available on-device JavaScript interpreter; JavaScriptCore is used on both Google Android and Apple iOS in the React Native framework [22]. Note that several JavaScript interpreters exist, and may differ between technical frameworks of the Interpreted approach.

5. Technical Implementations

5.1. Description

The purpose of the technical implementations is not to target large-scale enterprise problems, but rather to objectively understand and evaluate the approaches included. We have implemented a native module for React Native and a plugin for Cordova to communicate with the native-side of the two apps developed. All code listings presented throughout the article are shortened to a level where they explain their purpose, without taking up unnecessary space. Thus, code should not be expected to work if copied verbatim.

Both apps feature a simple user interface, including a button and an image preview (see figure 3). Upon button

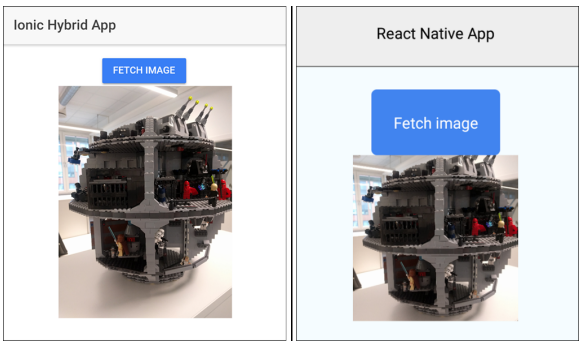

Figure 3. Hybrid Ionic app to the left, Interpreted React Native app to the right.

press, the app will send a predefined string, representing an image filename, through the native bridge to a function on the native-side which in turn will fetch the image from the device storage and return it in `Base64` back to the app-side, then display it in the image preview. For future reference, we call the feature *Fetch Image*. While the implementation itself is rather non-complex, it still illustrates how two-way bridge communication function together with a native call to the device's file storage.

5.2. Implementations

5.2.1. Hybrid. Figure 4 illustrates the overall code architecture and flow of data between app-side and native-side in the Cordova app and plugin. In this section, we elaborate on different parts of the illustration, using code for explanation and reasoning.

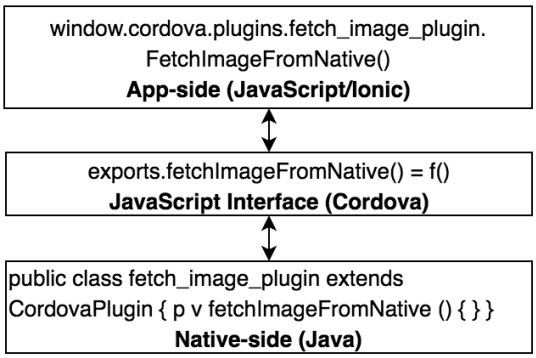

Figure 4. Cordova plugin flow.

An Apache-maintained Command Line Interface (CLI) tool was available for scaffolding new Cordova plugins, named *Plugman*. Using Plugman, we generated a new plugin named `fetch_image_plugin`. The CLI generated a folder structure with some boilerplate code to get started. The file `fetch_image_plugin.js` contains the JavaScript bridge. Cordova registers the method automatically as we export it. The method accepts a filename (path)

and two callbacks. Any values returned from the native side will be returned through the callbacks, whether being, in our case, a `Base64` string or an error message. Cordova exposes an `exec()` method which acts as a bridge abstraction between the app-side and the native-side of the app:

Listing 1. JavaScript Cordova Interface Code

```
var exec = require('cordova/exec');
exports.fetchImageFromNative = function(
    fileName, successCB, errorCB) {
        exec(successCB, errorCB,
          "fetch_image_plugin",
          "fetchImageFromNative",
          [fileName] );
};
```

The `src/` folder contains folders for Android and iOS. The file embodied in both folders contain platform-specific code which Cordova communicates with using the code from Listing 1. Listing 2 is a shortened version of the code from `fetch_image_plugin.java`. The class name `fetch_image_plugin` matches the string from Listing 1, and so does the `action` parameter. It calls `fetchImageFromNative`, locating the image file on the device, converts it to `Base64` and sends it to the app-side using a callback.

Listing 2. Native-Side Java Plugin Code

```
fetch_image_plugin extends Cordova {
  execute(action, file, cb) {
    this.fetchImageFromNative(file, cb);
  }
  fetchImageFromNative(file, cb) {
    File imgFile = new File(file);
    byte[] arr = FileUtils.
        readFileToByteArray(imgFile);
    String file64 = Base64.
        encodeToString(fileByteArray);
    cb.success(file64);
  }
}
```

In the Hybrid app, we locate the plugin and method `fetch_image_plugin.fetchImageFromNative` from listing 1 as part of the global `window.cordova` object. We pass the filename and two callbacks function, the latter returning the image in `Base64` or an error:

Listing 3. JavaScript/TypeScript Cordova Execution

```
window.cordova
  .plugins.fetch_image_plugin
  .fetchImageFromNative(
    "fileName.png",
    (imageB64: string) => { .. },
    (err: any) => { .. }
);
```

This section illustrates in code how Cordova (thus Hybrid apps) registers, executes and handles requests from the app-side to the native-side, how it communicates with and fetches a file from the device file storage, and in the end returns a `Base64` string back to the app-side.

5.2.2. Interpreted. As further elaborated upon in the discussion section, no standardised library or patterns are available for Interpreted apps in the way Cordova is available for Hybrid apps. This results in a variety of different bridging implementations. React Native has its own, opinionated implementation, which we further explore.

In React Native, *native modules* are used for bridging app-side and native-side. These modules are developed fully in the platform-specific programming language corresponding to the platform the module communicates with. We developed the *Fetch Image* module for Android, thus used Java for the implementation.

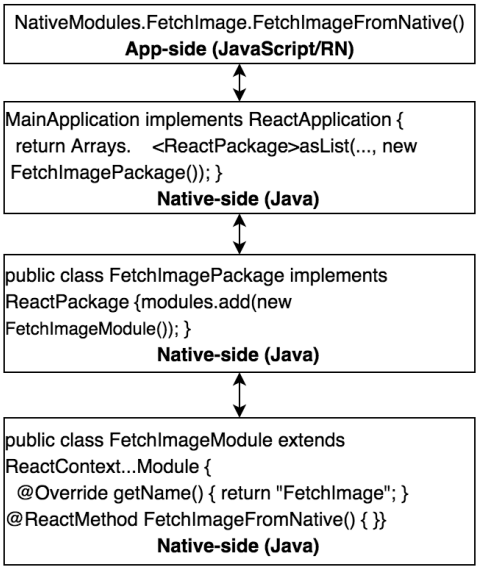

Figure 5. React Native module flow.

Figure 5 illustrates the overall code architecture and flow of data between app-side and native-side in the custom native module. In this section, we dive into the different parts of the illustration, using code for explanation and reasoning.

We started by building out the necessary files, structure and boilerplate code needed. All module-specific logic was placed in a single file, `src/../fetchimage/FetchImageModule.java`. The `getName()` method tells the module system under which name it should register the module (see listing 7), in our case it is reachable as `FetchImage`. By annotating native-side methods with `@ReactMethod`, React Native is able to register and expose them through the bridge. As displayed in the listing below, the business logic for reading

from device storage and encoding the file into `Base64` is identical to that in the Hybrid app.

Listing 4. React Native Java Module Code

```
FetchImageModule extends React {
  getName() { return "FetchImage"; }

  FetchImageFromNative(file, cb) {
    File imgFile = new File(file);
    byte[] arr = FileUtils.
        readFileToByteArray(imgFile);
    String file64 = Base64.
        encodeToString(fileByteArray);
    cb.invoke(file64);
  }
}
```

The module package file is located at `src/../fetchimage/FetchImagePackage.java` and contains boilerplate code for registering the module itself to the module system. This makes the module available from the app-side JavaScript environment.

Listing 5. React Native Java Package Code

```
FetchImagePackage implements React {
  createNativeModules() {
    List<NativeModule> modules = ...;
    modules.add(new FetchImageModule());
    return modules;
  }
}
```

Finally, `src/../MainApplication.java` notifies React Native the module's existence. We add `FetchImagePackage` to the array of React packages.

Listing 6. React Native Java Application Code

```
MainApplication implements ReactApp {
  getPackages() {
    return Arrays.asList(...,
        FetchImagePackage());
  }
}
```

From the app-side, we import the `NativeModules` registry into the index.android.js file. The module, `FetchImage`, and its method, `FetchImageFromNative` are available as part of the NativeModules registry, and is used as illustrated in listing 7. The module name comes from listing 4 and the `getName()` method. In accordance to the `FetchImageFromNative` method signature from listing 4, we pass a filename (path) and two JavaScript functions (callbacks).

Listing 7. JavaScript React Native Execution

```
import {NativeModules} from 'react-
    native';

NativeModules
  .FetchImage
  .FetchImageFromNative(
    "fileName.png",
    (imageB64) => { .. },
    (err) => { .. }
);
```

This section illustrates in code how React Native, thus one framework of the Interpreted approach, registers, executes and handles requests from the app-side to the native-side, how it communicates with and fetches a file from the device storage, and in the end returns a `Base64` string back to the app-side.

6. Preliminary Performance Results

To provide an empirical foundation for future performance-oriented research, we scrutinized both bridges' execution time for the purpose of comparison and discussion. As the implementations are JavaScript-based, both environments had access to the `performance.now()` API function. By calling the function immediately before executing the code in Listing 3 (Ionic) and 7 (React Native), and immediately inside the success callbacks, we could calculate how performant each framework were at executing bridge communication from the app-side, having the native-side fetch the requested file from the device's disk, then return the file in Base64 format back up the bridge into the app-side.

The method used involved clicking the *Fetch Image* button (as seen in Figure 3) in each app 10 times to account for variations. As for our results, we found that the average execution time for the Cordova-based Ionic app was `212.81ms`, while the React Native app executed the function at an average time of `1156.85ms`. Thus, the Hybrid app was more than five times faster than the Interpreted app regarding communication between the app-side and native-side.

The tests were conducted on an LG Nexus 5X running Android 7.1.2 *Nougat*. Future research should include a variety of devices for greater generalization and validity.

7. Discussion

A fundamental difference between Interpreted and Hybrid frameworks is that while Hybrid usually build on Cordova, Interpreted frameworks have no such standardized library to build upon. This results in great fragmentation of bridging implementations and techniques, each unique to the

framework it belongs to. Notable mentions of Interpreted frameworks includes React Native, NativeScript, Fuse and Titanium Appcelerator. While they all belong to the same approach, their underlying implementations of app-side native-side communication, or bridging, differs greatly.

Nevertheless, improvements and progression for cross-platform development has been noted when comparing state of the art against previous research. Findings presented by Corral *et al.* [23] (2012) indicated that cross-platform approaches suffered from lack of device feature access. Also Xanthopoulos and Xinogalos [10] (2013) claimed that Interpreted and Hybrid apps' access to native device feature APIs is *limited*. Lachgar and Abdali [24] (2017) considers *"access device-specific features"* to be of Medium level for Hybrid apps, while native scoring High, web scoring Low. As such, it is interesting that, based on our presented research, we are unable to identify features that cannot be bridged via plugins or native modules. Thus, we would consider the level of access to device features in the Hybrid and Interpreted approaches to be similar to the native approach.

In terms of our implementations, the Hybrid app's Cordova plugin required less boilerplate code than the Interpreted app's native module (ref. previous code listings and figures 4 and 5). Using the Plugman CLI tool for generating the required file and folder structure for the Cordova plugin, the majority of the plugin development was targeted towards writing business logic rather than boilerplate.

Development of the native module in React Native was more time consuming, where stitching together the different pieces of bridge and module led to more cognitive overhead. The business logic for device storage file retrieval and `Base64` encoding was simply Java for Android, and worked regardless of approach. As such, an important finding is how converting an existing cross-platform app's custom plugins/native modules from one approach to another would mean mostly changing non-business logic, i.e. framework-required boilerplate code.

The results from our preliminary performance tests, as presented in Section 6, indicate that bridges in Cordova are more than five times faster to return a result from the native-side, compared to the native module in React Native. These findings can be of even greater importance for more complex or heavy computing operations. However, framework-level or app-level optimizations may be able to decrease the difference.

As previously mentioned, our upcoming survey questionnaire study will provide insights on issues the industry relate to cross-platform development. The use of bridges and native functionality in cross-platform apps, as presented throughout this current paper, was through the survey identified as a major issue. This functionality is also mentioned as a *requirement* for cross-platform frameworks by Gaouar *et al.* [6] and Heitkötter *et al.* [2]. Thus, the importance of our technical research and preliminary performance testing could be acknowledged by the presence of the issue throughout previous research as well as in the developer communities.

Both figures 4 and 5 illustrates the level of abstraction provided by both the frameworks and approaches, and the ease of implementing native-side methods and functions reachable from the app-side codebase via bridges. Where the Cordova plugin required less boilerplate and could be generated using a CLI tool, the React Native module did not require an extensive amount of code to get started either. We found both approaches to properly facilitate the use of native-side functions and device APIs in cross-platform development, while at the same time be less complex than what the upcoming survey results illustrate. If we were to suggest the optimal approach based on our context and research results, the Hybrid (Cordova) implementation better facilitated the creation and development of custom plugins while also executing bridge communication more than five times faster than the Interpreted approach.

8. Conclusion and Future Work

This study present an implementation-oriented comparison of two cross-platform approaches, specifically Hybrid and Interpreted, and how they facilitate communication between app-side and native-side, as well as the use of native functionality, including device file storage retrieval. One major difference between the two approaches is Hybrid's dependency on a WebView component, whereas the Interpreted approach handles bridging rather through the use of on-device interpreters. Hybrid's WebView is an abstraction layer for native-side calls to be passed through, while also being responsible for rendering app content.

As our implementations illustrate, bridging native features such as communication with the device's file system, are not complex nor tedious programming tasks. How it's done, however, depends on approach and technical framework. The functionality included in the presented implementations, both the React Native module and the Cordova plugin, were simple to implement regardless of required boilerplate code. Both code and execution flow were easy to reason about, as displayed throughout the figures and code listings. Thus, communication between app-side and native-side in both chosen approaches did not present any clear issues, and the frameworks used did well to facilitate plugin development. The Hybrid-based Cordova app did however require less boilerplate code, and proved to be five times faster than the Interpreted app at communicating back and fourth between the app-side and the native-side.

This, in turn, answers our research question *"How programmatically complex is the development and use of communication bridges allowing for access to platform-specific device features in Hybrid and Interpreted apps?"*.

For future research, we aim to investigate differences between Interpreted approach frameworks, as suggested by

Heitkötter *et al.* [2]. These frameworks rely on their own implementation of bridges, as no standard has yet been defined. We will build on the method and data presented in Section 6, and design a comprehensive study targeting performance differences between such frameworks, drawing from previous performance-oriented research such as [14]. Additionally, we will further investigate other topics arising from the upcoming survey questionnaire study mentioned.

References

[1] T. Majchrzak, A. Biørn-Hansen, and T.-M. Grønli, "Comprehensive analysis of innovative Cross-Platform app development frameworks," in *Proceedings of the 50th Hawaii International Conference on System Sciences*. scholarspace.manoa.hawaii.edu, 2017, pp. 6162–6171.

[2] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, "Evaluating Cross-Platform development approaches for mobile applications," in *Web Information Systems and Technologies*, ser. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 18 Apr. 2012, pp. 120–138.

[3] H. Heitkötter, T. A. Majchrzak, and H. Kuchen, "Cross-platform model-driven development of mobile applications with md2," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 18 Mar. 2013, pp. 526–533.

[4] C. Escoffier and P. Lalanda, "Managing the heterogeneity and dynamism in hybrid mobile applications," in *2015 IEEE International Conference on Services Computing*. IEEE, Jun. 2015, pp. 74–81. [Online]. Available: http://dx.doi.org/10.1109/SCC.2015.20

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

[6] L. Gaouar, A. Benamar, and F. T. Bendimerad, "Desirable requirements of cross platform mobile development tools," *Electronic Devices*, vol. 5, pp. 14–22, Mar. 2016.

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

[8] J. Perchat, M. Desertot, and S. Lecomte, "Component based framework to create mobile cross-platform applications," in *Procedia Computer Science*, vol. 19. ScienceDirect, Jun. 2013, pp. 1004–1011.

[9] I. T. Mercado, N. Munaiah, and A. Meneely, "The impact of cross-platform development approaches for mobile applications from the user's perspective," in *Proceedings of the International Workshop on App Market Analytics*, ser. WAMA 2016. New York, NY, USA: ACM, 14 Nov. 2016, pp. 43–49.

[10] S. Xanthopoulos and S. Xinogalos, "A comparative analysis of cross-platform development approaches for mobile applications," in *Proceedings of the 6th Balkan Conference in Informatics*, ser. BCI '13. ACM, 2013, pp. 213–220.

[11] H. Heitkötter, S. Hanschke, and T. A. Majchrzak, "Comparing cross-platform development approaches for mobile applications," in *Proceedings 8th WEBIST*. SciTePress, Apr. 2012, pp. 299–311.

[12] T. Y. Adinugroho, Reina, and J. B. Gautama, "Review of multi-platform mobile application development using WebView: Learning management system on mobile platform," in *Procedia Computer Science*, vol. 59. Elsevier, 2015, pp. 291–297.

[13] A. Biørn-Hansen, T. A. Majchrzak, and T.-M. Grønli, "Progressive web apps: The possible web-native unifier for mobile development," in *Proceedings of the 13th International Conference on Web Information Systems and Technologies*, T. A. Majchrzak, P. Traverso, K.-H. Krempels, and V. Monfort, Eds. SCITEPRESS, Apr. 2017, pp. 344–351. [Online]. Available: http://www.scitepress.org/DigitalLibrary/PublicationsDetail.aspx?ID=MUwJbNP+h98=&t=1

[14] M. Ciman and O. Gaggi, "An empirical analysis of energy consumption of cross-platform frameworks for mobile development," *Pervasive and Mobile Computing*, 26 Oct. 2016.

[15] A. Puder, N. Tillmann, and M. Moskal, "Exposing native device APIs to web apps," in *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*. ACM, 2 Jun. 2014, pp. 18–26.

[16] C. Bouras, A. Papazois, and N. Stasinos, "A framework for Cross-Platform mobile web applications using HTML5," in *2014 International Conference on Future Internet of Things and Cloud*. IEEE, Aug. 2014, pp. 420–424. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6984231

[17] N. Gok and N. Khanna, *Building Hybrid Android Apps with Java and JavaScript*. O'Reilly Media, Incorporated, 2013.

[18] A. Ribeiro and A. R. da Silva, "Survey on Cross-Platforms and languages for mobile apps," in *2012 Eighth International Conference on the Quality of Information and Communications Technology*, 2012, pp. 255–260. [Online]. Available: http://dx.doi.org/10.1109/QUATIC.2012.56

[19] Cordova, "Platform support - documentation," https://cordova.apache.org/docs/en/latest/guide/support/index.html#core-plugin-apis, 14 Jun. 2016, accessed: 2017-2-2.

[20] Cordova Plugins, "Plugin search - apache cordova," https://cordova.apache.org/plugins/, accessed: 2017-2-6.

[21] L. Delia, N. Galdamez, P. Thomas, L. Corbalan, and P. Pesado, "Multi-platform mobile application development analysis," in *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*, May 2015, pp. 181–186.

[22] React Native, "JavaScript environment," https://facebook.github.io/react-native/docs/javascript-environment.html, 1 Feb. 2017, accessed: 2017-2-6.

[23] L. Corral, A. Janes, and T. Remencius, "Potential advantages and disadvantages of multiplatform development Frameworks–A vision on mobile environments," in *Procedia Computer Science*, vol. 10. SciVerse ScienceDirect, 9 Aug. 2012, pp. 1202–1207.

[24] M. Lachgar and A. Abdali, "Decision framework for mobile development methods," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 2, pp. 110–118, 2017. [Online]. Available: http://thesai.org/Publications/ViewPaper?Volume=8&Issue=2&Code=ijacsa&SerialNo=15