Kraken 2.0: A platform-agnostic and cross-device interaction testing tool

William Ravelo-Méndez, Camilo Escobar-Velásquez[1], Mario Linares-Vásquez[1]
Universidad de los Andes, Bogotá, Colombia
{wr.ravelo, ca.escobar2434, m.linaresv}@uniandes.edu.co

***Abstract*—Mobile devices and apps have a primordial role in daily life, and both have supported daily activities that involve humans interaction. Nevertheless, this interaction can occur between users in different platforms (e.g., web and mobile) and devices. Because of this, developers are required to test combinations of heterogeneous interactions to ensure a correct behavior of multi-device and multi-platform apps. Unfortunately, to the best of our knowledge, there is no existing open source tool that enables testing for those cases. In this paper, we present an improved version of our tool KrakenMobile, an open source tool that enables the execution of interactive End-2-End tests between Android devices. This new version, Kraken 2.0, has new capabilities such as execution of platform-agnostic interactive End-2-End tests (e.g., web and mobile), and has been migrated from Ruby to NodeJS to improve its usability. Kraken2.0 is publicly available on GitHub (https://bit.ly/3oKPFcv). Videos: https://bit.ly/3fIfRXa**

***Index Terms*—Signaling; Automated Testing; Cross-device Testing; Cross-application Testing**

I. Introduction

Interaction with devices that have computing capabilities has become part of daily life tasks; it includes the usage of applications that are executed on desktops and laptops as well as mobile devices such as phones and tablets. Along with this plethora of available devices, users also have different options concerning the platforms used for executing and distributing applications, *e.g.,* web and mobile versions of the same app.

The availability of different app execution environment, have exerted a pressure over developers since users expect high quality interactions between their different devices when using an app. Additionally, developers must ensure the correct behavior of apps in scenarios where users interact on different platforms. An example of this are social media apps, that allow users to post content at web browsers and interact with it from a mobile device, or in the other way around (*i.e.,* mobile then web). Therefore, developers and practitioners need to test not only application scenarios executed on a specific platform but also interaction between users from different platforms or devices, and interactions of the same user but on different platforms. However, while testing of web and mobile applications are research fields that have proposed a plethora of approaches and tools, no current publicly available tool allows for creating testing scenarios that are executed on different platforms in an interactive way; automated testing of apps that provide cross-platform interactions is not a well explored field yet.

An example of this behavior can be an email communication between two people; at the beginning of the interaction both of them are using their browser to exchange emails, and at some point one person needs to change into the email android app, so she writes back from her cellphone. After answering a couple of the emails, the user that was still using the web browser decides to continue later on the mobile app, but she decides first to write a draft of its last email so he can finish it from its cellphone. At this point, these interactions have required different versions of an app to be capable of interact between them. Additionally, it has depicted not only interaction between different users, but also the interaction of the different environments of the same app . This type of interactions are common between people, but in order to test them, nowadays, practitioners and developers need to mock the interaction since there is no available approach that supports multi platform/device testing.

In this paper we present **`Kraken v2.0`**, an open source automated End-2-End (E2E) testing tool for defining and executing scenarios that involve inter-communication between two or more browsers or mobile devices. **`Kraken v2.0`** is an enhanced version of **Kraken v1.0** [1], which (i) includes fixes to issues, (ii) improves user experience, and (iii) has new features to support web-mobile interaction and fuzzing. In order to show **`Kraken v2.0`** capabilities, we used the tool with 10 combinations of web and mobile apps, in which we created and executed cross-device E2E test scenarios. **`Kraken v2.0`** is publicly available on GitHub (https://bit.ly/3oKPFcv) and as a npm package: `kraken-node`.

II. Related Work

End-to-End testing (E2E) is defined as a technique to validate software quality at the system and acceptance levels by executing scenarios that combine several use cases. This implies that this testing technique does not restrict the test execution to a specific functionality and is more focused on complex test cases. For example, in an application for communication, an E2E scenario could include the following: a user must login, then she retrieves the list of contacts, opens a chat, and finally sends a message to reach a friend.

Several approaches and tools have been proposed for supporting E2E testing of web and mobile applications, but without enabling multi platform interaction. An example of the existing tools is Appium [2], that allows writing tests

[1]Escobar-Velásquez and Linares-Vásquez were partially supported by a Google Latin American Research Award 2018-2021.

for web or mobile apps, by using specialized frameworks that transform Selenium [3] commands into web and mobile specific commands. This enables E2E testing for both platforms. Nevertheless, it does not provide support for the interaction across different devices and platforms. Another existing approach is BrowserStack [4] that offers the execution of Selenium [3] tests in parallel in a user-defined grid of browsers. BrowserStack also supports the execution in parallel of Android app tests written using Appium [2] or Espresso [5]. This tool allows for writing tests that are device-agnostic, however, it does not enable communication between platforms to complete the interaction circle for a multi-platform app.

In the case of mobile apps, there are tools such as Calabash [6], UIAutomator [7], and Espresso [5]. These tools enable E2E for within Android apps, but all present downsides to solve the platform-agnostic environment presented previously. For example, UIAutomator is capable of performing tests that involve several apps within the same device. Nevertheless, the main app is treated as a blackbox, limiting the processing of the generated result. However, none of the mentioned approaches enables the interaction between android devices.

One might think that this interaction problem can be solved by using existing communication protocols such as Bluetooth, Server Sockets or even NFC. Nevertheless, these approaches introduce new limitations such as no support for offline-mode scenarios for internet-based protocols, as well as low consistency due to specific implementation of those protocols between devices and platforms.

To the best of our knowledge the only approach that enables interaction between devices is Octopus [8], a private solution created by the Uber Engineering team. Octopus, generates a communication channel between mobile devices by creating a message based interaction that are stored and handle by a orchestrator device. Despite Octopus proposes a solution to the inter-communication testing problem, it is not available for users outside of Uber, and works only for mobile apps.

III. The `Kraken v2.0` Tool

In this section we describe the main changes that **Kraken** undergo when going from its v1.0 to v2.0. The current version (2.0) can be considered as a new tool since it was migrated from ruby to typescript, leading to a set of changes such as the replacement of base test automation library, enhancement of the tool usage by easing its installation and operation, and extension of available features to provide the users with a more robust testing tool. However, **`Kraken v2.0`** maintains the original goal of the tool, enabling cross-device interaction-based testing of apps. After going through a robust experimentation with **Kraken v1.0** we identified the need to generate a new version to mitigate some limitations: First, because Ruby relies on UNIX tools, **Kraken v1.0** was particularly difficult to configure and use on Windows OS. Second, **Kraken v1.0** only allowed inter-communication between Android devices. Third, Calabash [6] stopped receiving support from the industry, leading to its deprecation since Android Oreo, which endangered also the compatibility of **Kraken v1.0**.

Table I: Capabilities comparison for the different tools

Feature	**Kraken1.0**	**Kraken2.0**
Parallel Execution	✓	✓
Signaling	✓	✓
Mobile device support	✓	✓
Web Browser support		✓
Cross-platform testing		✓
Generate random events over full- and part-of-screen	✓	✓
Kraken monkey	✓	✓
DataPool definition	✓	✓
Random Data generation		✓
Deployment to cloud (CI/CD)		✓
Extract snapshot of device		✓
Report generation	✓	✓
Test specification protocols	Gherkin	Gherkin, TS, JS
Step definition language	Ruby	TS, JS

Fourth, developers were interested in having more options for scenarios specification. To solve these issues we built the new version on top of WebdriverIO [9] and Appium [2], and combine them with the specification-by-example testing and signaling proposed by **Kraken v1.0** and Octopus [8]. Table I presents the differences between the two versions of the tool. **`Kraken v2.0`** includes additional capabilities such as: (i) tests execution with mobile and web interaction; (ii) new languages available for the specification of tests and steps; (iii) deployment in the cloud for CI/CD pipelines; (iv) on-demand extraction of UI snapshots of an android application at a given moment; (v) random data generation.

Fig. 1 presents the architecture of **`Kraken v2.0`**. It consists of several modules aimed at launching required devices/browsers, creating artifacts for each device/browser, supporting/orchestrating the signaling protocol, executing steps and generating reports. Components that support the new features depicted in Table I are highlighted by using a red border.

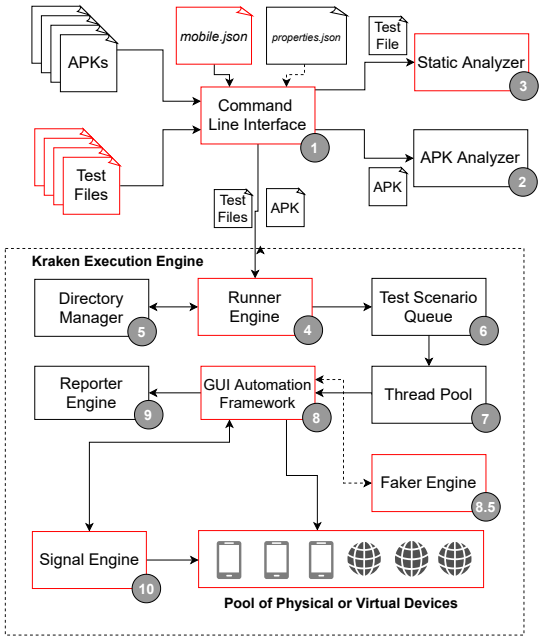

Fig. 1: The **`Kraken v2.0`** architecture and workflow.

A. Test Files and Static Analyzer

Originally, **Kraken v1.0** was built using Calabash [6] as framework for the test definition and execution. Because of this, the tests were required to be defined following a BDT-style specification. Additionally, in order to define custom test steps, the user must write them using ruby. As result of the migration to Appium, **Kraken v2.0** now allows users to additionally define tests using "native" typescript files. An example of a test written using typescript is presented in Snippet 1. With this new capability, the *Static Analyzer* component was also modified to be capable to check the compliance of the test files with the syntax provided by Appium and **Kraken v2.0**. It is worth noticing that Appium also supports BDT-style specifications, providing the users with backward compatibility of the existent test scripts created for *Kraken v1.0*.

Snippet 1: Kraken 2.0 scenario created with Typescript

```
1  let button = await client.$(`android=new UiSelector().resourceId
       ("es.usc.citius.servando.calendula:id/mi_button_skip")`)
2  await button.click();//
3
4  let burgerMenu = await client.$('//android.widget.ImageButton[
       @content-desc="Open"]');
5  await burgerMenu.click();//
6
7  let settings = await client.$(`android=new UiSelector().
       resourceId("es.usc.citius.servando.calendula:id/textView3
       ")`)
8  await settings.click();//
9
10 let back = await client.$('//android.widget.ImageButton[@content
       -desc="Navigate up"]')
11 await back.click();//
```

B. Cross-platform Interaction-based Multi-device testing

As part of the migration of the tool, **Kraken v2.0** now supports the definition and execution of tests over web browsers, since **Kraken v1.0** already performed signaling between devices, we extrapolated the signaling process to communicate two or more test execution processes regardless of the selected platforms. This was possible due to the capabilities provided by Appium, since the test definitions are platform-agnostic, and by using an internal framework **Kraken v2.0** translates the Selenium Webdriver commands into UIAutomator commands in the case of Android. Additionally, when it is used, Appium deploys an HTTP server that exposes a REST API that orchestrates the event transmission to the different devices and browsers.

C. Random Data Generation

In addition to the previously mentioned changes, **Kraken v2.0** allows for random data generation by using the FakerJS [10] library. This impacts the previous *(4) Runner Engine*, because when this mode is used within the test definition, a new task is created towards the *(8.5) Faker Engine* that is in charge of handling the library. This component generates the random data defined by the user and works as cache for the requested values in case a user decides to use them several times within the same test scenario. More detailed information regarding how to use this mode is presented in the following section.

IV. KRAKEN v2.0 IN ACTION

A. Console Interface

User interaction with **Kraken v2.0** is done via console commands. To start using **Kraken v2.0** it is only necessary to install the NPM package via the command `npm install kraken-node`; afterwards, users can (i) check if all prerequisites are fullfiled on the host machine, (ii) generate a test folder skeleton, and (iii) run the tests. Before running **Kraken v2.0** and depending on the type of devices that are going to be used, **Kraken v2.0** will require different prerequisites that are specified with the `kraken-node doctor` command; in addition, this command will display if the prerequisites are correctly configured as shown in Snippet 2.

Snippet 2: Kraken prerequisites

```
1  Checking dependencies...
2  Android SDK [Installed] (Required only for mobile testing -
       ANDROID_HOME)
3  Android AAPT [Installed] (Required only for Kraken's info
       command - ANDROID_HOME/build-tools)
4  Appium [Installed] (Required only for mobile testing)
5  Java [Installed] (JAVA_HOME)
6  Done.
```

After setting up the testing environment the first step to use **Kraken v2.0** is to generate a base project containing:

- A *.feature* test file (similar to the one in Snippet 3)
- A *.json* file where the information of the APK under test will be specified
- A web subfolder containing a javascript file for the step definition
- A mobile subfolder with files required for **Kraken v2.0** execution

To do this, the user should call the `kraken-node gen` command.

Snippet 3: Example test file

```
1  Feature: Example feature
2
3  @user1 @web
4  Scenario: As a first user I say hi to a second user
5  Given I navigate to page "https://www.google.com"
6  Then I send a signal to user 2 containing "hi"
7
8  @user2 @mobile
9  Scenario: As a second user I wait for user 1 to say hi
10 Given I wait for a signal containing "hi"
11 Then I wait
```

A test file should contain scenarios definition following a Gherkin+**Kraken v2.0** syntax. For example, the feature in Snippet 3 contains two scenarios, one per each device involved in the test. As it can be seen in lines 3 and 8 of Snippet 3, each scenario is linked to a specific user (*i.e.,* a device). The user tag follows a naming pattern: `@user(\d+)`. Also, each scenario can be executed on a different type of device such as a web browser or an Android device; to specify what type of device a scenario requires, the user should add the @web or @mobile tag.

Before executing the test and due to the usage of Appium as part of the architecture, **Kraken v2.0** requires the launcher activity name and package name of the APKs under test. This information must be specified in the *mobile.json* (Snippet 4) file at the root directory.

Snippet 4: mobile.json file

```
{
        "type": "singular",
        "apk_path": "<APK_PATH>",
        "apk_package": "<APK_PACKAGE>",
        "apk_launch_activity": "<APK_LAUNCH_ACTIVITY>"
}
```

In the case of specifying different APKs for each user then the tester should modify the *mobile.json* file to include the APK information of each user.

Snippet 5: mobile.json file for multiple APKs

```
{
        "type": "multiple",
        "@user1": {
                "apk_path": "<APK_PATH>",
                "apk_package": "<APK_PACKAGE>",
                "apk_launch_activity": "<APK_LAUNCH_ACTIVITY>"
        },
        "@user2": {
                "apk_path": "<APK_PATH>",
                "apk_package": "<APK_PACKAGE>",
                "apk_launch_activity": "<APK_LAUNCH_ACTIVITY>"
        }
}
```

When testing applications that are external to the organization or that the source code is not publicly available, it may happen that the APKs launcher activity name is not known for the tester, that is why **Kraken v2.0** offers the *apk-info* command that given an APK file path will retrieve this information by using the Android debug bridge (ADB). To run this command the tester should execute `kraken-node apk-file <APK_PATH>`, and **Kraken v2.0** will display information such as the one on Snippet 6

Snippet 6: APK info retrieved by the `kraken-node apk-file` command

```
Launch activity: es.usc.citius.servando.calendula.activities.
      StartActivity
Package: es.usc.citius.servando.calendula
```

Once the user has accessed to this information, she must update the *.feature* file provided by **Kraken v2.0**, to correctly configure the test execution. It is worth remembering that **Kraken v2.0** uses the Gherkin + **Kraken v2.0** syntax, thus, the tester can create new steps and use already defined Appium and WebdriverIO helper functions. The **Kraken v2.0** specific steps are:

1) Send signal: **Kraken v2.0** provides a step that sends a signal to a device. This step has two parameters: the user (*i.e.,* tag) that will receive the signal and the content of the signal. The structure of this step is:

Snippet 7: **Kraken v2.0** send signal

```
I send a signal to user (\d+) containing "([^\"]*)"
```

2) Read signal: This step can be used in two different ways: first, to set the expected content (see line 1 in Snippet 8); and third, to define the expected content and timeout (see line 2 in Snippet 8)

Snippet 8: **Kraken v2.0** wait signal

```
I wait for a signal containing "([^\"]*)"
I wait for a signal containing "([^\"]*)" for (\d+) seconds
```

3) Random Events Scenario: **Kraken v2.0** provides a scenario step that allows users to generate GUI-based random inputs by following the syntax presented in Snippet 9. This step can execute a given number of random events over the full screen (*line 1 in Snippet 9*) or on a specific region defined by the user as a portion of the screen (*line 2 in Snippet 9*).

Snippet 9: **Kraken v2.0** random step

```
I start a monkey with (\d+) events
I start a monkey with (\d+) events from height (\d+)% to (\d+)%
      and width (\d+)% to (\d+)%
```

4) Kraken Monkey: **Kraken v2.0** introduces another step that also sends and reads signals randomly, in addition to other input events (*e.g.,* text input, tap, etc.). The structure of this step is: `I start a kraken monkey with (\d+)events`

5) On-demand snapshot extraction: **Kraken v2.0** allows users to extract a GUI snapshot of Android applications by using a predefined step. This feature can be used by practitioners to understand the UI components hierarchy (along with its content). The structure of this step is: `I save device snapshot in file with path "([^\"]*)"`

6) Properties file: **Kraken v2.0** uses properties files to store data such as passwords or API keys that should be used in your test cases. A properties file should be a manually created JSON file with the structure presented in the following snippet:

```
{
    "@user1": {
        "PASSWORD": "test"
    }
}
```

Note that the key-value pairs are organized by user tags. In order to use the values defined in the properties file, the property should be invoked in a *.feature* file using "< ... >" as in the following: `I see the text "<PASSWORD>"`.

Finally, after writing the scenarios, the last step for using **Kraken v2.0** is running it with the `kraken-node run` command; it must be called in the folder that contains the *.feature* files. In case the user wants to use a property file, then the command must include the flag `--properties=<properties_path>`.

B. Fuzzing

Kraken v2.0 offers a fake string generator thanks to the library FakerJS; the list of supported faker types are: *Name*, *Number*, *Email*, *String*, *String Date*.

Kraken v2.0 keeps a record of every fake string generated; to this, each string must have an id. To generate a Faker string you need to follow the structure "$FAKERTYPEs_ID" as presented in the following example:

```
@user1
Scenario: As a user
    Then I enter text "$name_1" into field with id "view"
    Then I enter text "$date_1" into field with id "form_date"
```

As mentioned before, **Kraken v2.0** keeps record of every string generated with an given id, this provides users with the possibility of reusing this string later in test scenarios. To reuse a string, the user needs to append a $ character to the fake string as in line 6 of the following example:

```
@user1
Scenario: As a user
        Given I wait
        Then I enter text "$name_1" into field with id "view"
        Then I press "add_button"
        Then I should see "$$name_1"
```

C. Web Reports

When finishing the execution of the test files, **Kraken v2.0** will generate a directory (in the current folder) that contains three reports: (i) general report, (ii) execution report by device, and (iii) execution detail by device.

Source code, examples, more detailed explanations of the commands, and videos showing **Kraken v2.0** on action are available at https://bit.ly/3oKPFcv.

D. Usage Examples

To show **Kraken v2.0** capabilities we created and executed test scenarios for 10 different apps that involved the interaction of two or more users/applications and also included web interaction. The capability of the framework to coordinate the signaling protocol between two or more devices running the same application is illustrated with (i) social and messaging applications such as AskFM or Facebook were one user shares information and then other users can access it; (ii) trivia games such as Kahoot where two or more users compete by answering a list of questions; and (iii) delivery apps such as Pibox were a delivery company requests a delivery guy for sending a package. Finally, we used **Kraken v2.0** to test scenarios that involved the coordination of two or more services by testing a scenario where one user plays a song with an audio streaming service app such as Spotify, and then another user listens to the song and recognizes it with a song identifier app such as Shazam.

One of the examples is presented as follows: Snippet 10 presents the steps for a test of an email app from a mobile app point of view (`user 1`). From lines 4 to 10, the user logins into the app, then in line 12 it starts composing and email, then from line 14 to 16 it fills out the email form, and finally the form is submitted (line 17). It is worth noticing that at line 20 the test sends a signal informing to `user 2` that the email was sent.

Complementary, Snippet 11 presents the steps to check the email in a browser (`user 2`). For this, in line 3 the test starts by visiting the email app home page, then from line 4 to 7 the scenario fills out the login form and enters the app. At this point, since no signaling related step has been executed yet, both scenarios will be executed in parallel, nevertheless, in line 8 of Snippet 11, the scenario specifies that `user 2` needs to wait for a signal with the label "`email-sent`", which was sent by `user 1` in line 20 of Snippet 10. This allows the test execution to understand where it should wait for the other users. Finally, at line 11 it checks if there is a new email with the subject used in Snippet 10.

Videos showing scenarios, and testing artifacts for the 10 apps are available at https://bit.ly/3Dz856j.

Snippet 10: Kraken 2.0 mobile scenario for email app

```
1  @user1 @mobile
2  Scenario: Send email from mobile app
3      Given I wait
4      When I click email input
5      When I enter text "kraken@gmail.com"
6      And I wait
7      And I click password input
8      And I enter text "kraken2020@"
9      And I wait
10     And I click on button with text "Login"
11     And I wait
12     And I click on button with text "Compose"
13     And I wait
14     And I enter text "krakentest2020@gmail.com" into field with
           id "destination_email"
15     And I enter text "Kraken test subject" into field with id "
           compose_subject_field"
16     And I click on screen 50% from the left and 50% from the top
17     And I enter text "My message"
18     And I press view with id "action_compose_send"
19     And I wait
20     Then I send a signal to user 2 containing "email-sent"
```

Snippet 11: Kraken 2.0 web scenario for email app

```
1  @user2 @web
2  Scenario: Check email from browser
3      Given I navigate to page "https://gmail.com"
4      When I enter "krakentest2020@gmail.com" into input field
           having id "identifierId"
5      And I click on element having id "identifiedNext"
6      And I enter "kraken2020@" into input field having css
           selector "#password > div.aCsJod.oJeWuf > div > div.
           Xb9hP > input"
7      And I click on element having id "passwordNext"
8      And I wait for a signal containing "email-sent" for 60
           seconds
9      And I navigate to page "https://gmail.com"
10     And I wait for 5 seconds
11     Then I should see text "Kraken test subject"
```

V. Conclusion & Future Work

We presented **Kraken v2.0**, the first publicly available tool for platform-agnostic cross-device interaction-based testing of Android and web apps. **Kraken v2.0** allows developers/testers to write testing scenarios in an E2E fashion for applications that require interaction of two or more users on different devices/browsers and between platforms. **Kraken v2.0** also allows developers/testers to combine E2E scenarios with random generation of events and data. Future work will be devoted to (i) support the execution of scenarios in iOS devices, (ii) include a systematic exploration mode to enable cross-device GUI ripping, and (iii) conduct more experiments of the tool with practitioners.

References

[1] W. Ravelo-Méndez, C. Escobar-Velásquez, and M. Linares-Vásquez, "Kraken-mobile: Cross-device interaction-based testing of android apps," in *ICSME'19*.

[2] J. Foundation. Appium. [Online]. Available: http://appium.io/

[3] SeleniumHQ. Selenium. [Online]. Available: https://www.selenium.dev/

[4] BrowserStack. Browserstack. [Online]. Available: https://www.browserstack.com

[5] Android. (2019) Espresso. [Online]. Available: https://bit.ly/3FvQcpK

[6] Calabash. (2019) Calabash-android. [Online]. Available: https://github.com/calabash

[7] Android. (2019) Ui automator. [Online]. Available: https://bit.ly/3DxcSW1

[8] B. J. A. Chow. (2015) Octopus to the rescue: The fascinating world of inter-app communications at uber engineering. [Online]. Available: https://eng.uber.com/rescued-by-octopus/

[9] O. foundation. Webdriver.io. [Online]. Available: https://webdriver.io/

[10] Marak. faker.js. [Online]. Available: https://bit.ly/30KyBeX