Testing Nearby Peer-to-Peer Mobile Apps at Large

Lakhdar Meftah, Romain Rouvoy, Isabelle Chrisment

Testing Nearby Peer-to-Peer Mobile Apps at Large

Lakhdar Meftah *, Romain Rouvoy ‡ and Isabelle Chrisment §
* Inria / University of Lille, France
‡ University of Lille / Inria / Institut Universitaire de France, France
§ Telecom Nancy / Inria, France

***Abstract*—While mobile devices are widely adopted across the population, most of their remote interactions usually go through Internet. However, this indirect interaction model can be assumed as inefficient and sensitive when considering communications with neighboring devices. To leverage such weaknesses, nearby *peer-to-peer* (P2P) communications are now included in mobile devices to enable device-to-device communications over standard wireless technologies (WiFi, Bluetooth). While this capability supports the design of collaborative whiteboards, local multiplayer gaming, multi-screen gaming, offline file transfers and many other applications, mobile apps using P2P are still suffering from app crashes, battery issues, and bad user reviews and ranking in app stores. We believe that this lack of quality can be partly attributed to the lack of tool support for testing P2P mobile apps at large. In this paper, we therefore introduce a test framework that allows developers to implement reproducible testing of nearby P2P apps. Beyond the identification of P2P-related errors, our approach also helps to tune the discovery protocol settings to optimize the deployment of P2P apps.**

***Index Terms*—Integration Test, Mobile apps, Peer-to-peer communications**

I. Introduction

Mobile devices are intensively used to support different forms of interactions among users in the physical world. However, most of these interactions are currently conveyed through remote communications over the Internet, even when the involved peers are co-located. This indirect communication scheme may suffer from communication bottlenecks depending on the *Quality of Service* (QoS) of the underlying network, as well as disclosure of sensitive data by relying on third parties (*e.g.*, public local area network).

The Google Nearby framework has been released in 2014 to fulfill such weaknesses and to deliver a native support to enable *peer-to-peer* (P2P) communications among neighboring devices. In particular, Google Nearby supports device-to-device exchanges by implementing network advertisement, discovery and communication features atop of standard wireless technologies (including WiFi hotspots, Bluetooth, BLE). Unlike previous *ad-hoc* communication APIs, users are not prompted to turn on Bluetooth or WiFi—Google Nearby enables these features when required, and restores the device to its prior state once the app is done using the API—thus ensuring a smooth user experience. Google Nearby, therefore, supports the design of collaborative whiteboards, local multiplayer gaming, multi-screen gaming, offline file transfers and many other applications.

However, 4 years after releasing this framework, only a limited number of Android apps have integrated Google Nearby in production (only 272 apps according to `42matters.com` when we wrote this paper). We guess that this lack of adoption is probably due to the difficulty to properly configure and test *peer-to-peer* mobile apps. Moreover, we also observed that users of *peer-to-peer* mobile apps tend to complain from app crashes and battery issues, which inevitably impacts the adoption of these apps at large.

This paper, therefore, addresses this limitation and promotes new testing abstraction to enhance the development of *peer-to-peer* mobile apps based on the Google Nearby framework. Because faithfully reproducing testing conditions with physical devices is extremely hard for developers, our solution—named PeerFleet—delivers an alpha testing environment to assess the behavior of their *peer-to-peer* mobile apps at large and tune the associated discovery parameters. More specifically, PeerFleet can control a crowd of device emulators through specific *actions* and check the expected behavior of mobile apps through the definition of specific *assertions*.

The remainder of this paper is therefore organized as follows. We motivate this work by introducing the Google Nearby technology and reporting on some empirical evaluation of its adoption in the wild in Section II. Section III introduces and discusses how *peer-to-peer* mobile app based on Google Nearby can be effectively tested by state-of-the-art frameworks. Based on our observation, we introduce our PeerFleet framework in Section IV, which is then evaluated in Section V. Then, we show the results of our experimental validation in Section VI. We discuss related works in Section VII before concluding in Section VIII.

II. Background & Motivations

A. Overview of the Google Nearby Framework

Google Nearby is a broad framework developed by Google to support the development of pervasive, peer-to-peer, mobile apps by leveraging the diversity of communication interfaces made available by mobile devices. This framework supports not only the discovery of peers, but also communication capabilities and interoperability with Apple iOS.

Android Wi-Fi Peer-to-Peer. The legacy *Android Wi-Fi peer-to-peer* framework includes a support for both *peer* discovery and *service* discovery.[1] This core framework therefore provides the primitives to discover and pair two devices, by exposing IP addresses for both endpoints. However, beyond this base support, developers are required to develop their

[1] https://developer.android.com/guide/topics/connectivity/wifip2p

TABLE I
50^{st} NEARBY APPS RATINGS IN THE GOOGLE PLAY STORE (APRIL 2018)

App package name	Rating	Bad ratings	WiFi Direct issues	*Not working*	*Crashing*	*Battery*
com.remaller.android.wifitalkie_lite	4.2	352	1	6	1	-
com.elmoha.Wifitalkie	3.8	174	5	16	1	-
org.servalproject	4.2	175	7	21	6	1
com.opengarden.firechat	3.6	9,592	2	21	15	2
com.offlinechatapp.android	3.7	118	25	-	15	-
com.estmob.android.sendanywhere	4.7	4,019	-	3	-	-
com.everwasproductions.demo.wifidirect	3.5	275	-	13	1	-
ru.kolif.wffs	4.5	28	-	1	-	-
com.ftp.WifiDirectFileTransfer	3.5	106	28	-	1	-
ca.nickadams.wifi.direct.file.transfer	3.1	118	2	10	6	-
aaqib.shareonwifi	4.1	11	-	1	-	-

own application protocol (*e.g.*, by exposing a TCP server) to exchange data across multiple devices, which often leads to errors and makes the testing of these apps particularly hard.

Google Nearby Connections. To leverage this issue, Google developers recently released a Nearby framework,[2] which is integrated into any Android devices via Google Play Services. The *Google Nearby Connections* delivers an offline peer-to-peer communication layer without the need for a cloud provider or a WiFi router. The connection is achieved as follows: whenever a device discovers services advertised by other devices (printer, beacon, etc.), it automatically connects to one of them and—once connected—the two devices can exchange data through `Payload` objects, which can convey bytes, files or streams. While the *Android Wi-Fi peer-to-peer* framework handles creating a connection between nearby devices, *Google Nearby Connections* adds another layer for handling data transfer between nearby devices, which raises the communication abstractions and eases the test of the mobile apps using this framework.

Google Nearby Messages. The *Google Nearby Messages* extends the Nearby framework to support both Android and iOS devices. In this mode, while device discovery is supported by hardware interfaces (WiFi, Bluetooth, etc.), the communication is achieved through an Internet server for small payloads size. However, this API can only be used for small payloads size (from 3 KB to 100 KB) because it passes by a cloud server, which imposes a daily quota. Usually, it is used to pair different devices by exchanging ids, then the file exchange can be done without Internet.

iOS Multipeer Connectivity. The iOS MultipeerConnectivity framework provides a support to discover nearby services and connect with them.[3] The connection is completed in two phases, first by discovering nearby peers, and then by initiating a session between the peers, the session also handles sending payloads between peers.

B. Presence of Google Nearby in the Play Store

Table I reports on the ratings and issues raised by the users of peer-to-peer Android apps. These apps were selected from the Google Play store by querying the *WiFi Direct* keyword, which is one of the technology supported by Google Nearby. The *bad ratings* column refers to the number of times the app obtained the lowest rating—*i.e.*, 1 out of 5 stars. By processing user reviews, we observed that the users are complaining about the app crashing, the app is not working and on WiFi Direct issues. Examples of complaints mentioned in reviews are:

- *Keeps asking for WiFi On/Off Permission every single time I unlock my device.*
- *Stops after startup if I have any internet connection.*
- *Turns wifi off and on even when not using it.*
- *Unstable & drop wifi speed when running app background!!!!*

Given the feedbacks reported by the app users, one can guess that the developed mobile apps could benefit from a more advanced testing support to anticipate and isolate the situations reported online in order to improve the quality of experience of their users and avoid bad ratings that severely impact their promotion.

Similarly, the Thali project[4] is a Cordova Plugin that brings GOOGLE NEARBY P2P communications to Cordova mobile apps. In particular, they clearly report that *"Getting peer to peer working on Android has been and continues to be a heck of a challenge."* They discuss the issues they had got using nearby P2P, especially with Wi-Fi Direct Service Discovery API.[5]

Finally, Table II reports on the 10 most downloaded apps that are using the GOOGLE NEARBY Connections API.[6] We observed that these apps failed to update to the latest version of the API, although this version proposes some major improvements. The reason behind this may be the fact that migrating to the new version will take too much time, and they will have to walk through testing the app again to get a stable version without any support from the testing frameworks.

[2]https://developers.google.com/nearby

[3]https://developer.apple.com/documentation/multipeerconnectivity

[4]http://thaliproject.org

[5]http://thaliproject.org/androidWirelessIssues

[6]Obtained from 42matters.com analytical website, then using *apkanalyser* to filter the APKs depending on the `ConnectionsClient` class from Google Nearby Connections, which unavailable in the latest versions of the framework.

TABLE II
APPS IN THE GOOGLE PLAY APP STORE USING THE GOOGLE NEARBY CONNECTIONS API

App (package name)	downloads
WhatsApp Messenger (com.whatsapp)	1,000,000,000+
WeChat (com.tencent.mm)	100,000,000+
BBM - Free Calls & Messages (com.bbm)	100,000,000+
Hola Free VPN Proxy (org.hola)	50,000,000+
Email App for Mail.Ru (ru.mail.mailapp)	50,000,000+
AirDroid (com.sand.airdroid)	10,000,000+
XShare (com.infinix.xshare)	10,000,000+
iPair-Meet (com.ipart.android)	5,000,000+
DataBot IA (com.testa.databot)	1,000,000+
FITAPP (com.fitapp)	1,000,000+

Conclusion. Overall, one can observe that Google Nearby is adopted by the apps published in the Play store, including some major ones like WhatsApp Messenger, WeChat or Hola VPN. Nonetheless, most of these apps seem to suffer from bad reviews, some pointing the weaknesses of the integration of Google Nearby. Furthermore, we also observed that the most famous apps integrating Google Nearby did not migrate to the latest version of the API, thus potentially exposing their users to security leaks and quality of experience degradations.[7]

We therefore think that the support for Google Nearby can be strengthened by extending existing development tools and methodologies to help Android developers better integrate and test peer-to-peer mobile apps, which remains a tedious task so far.

III. STATE OF THE ART IN MOBILE APP TESTING

In this section, we consider state-of-the-art approaches and frameworks that can be considered to test nearby apps. Then, we report on the challenges and opportunities that are motivated by the limitations of actual practices.

A. State of Practice for Testing Nearby Mobile Apps

Nearby apps are standard mobile apps leveraging an API provided by the Android SDK. As such, one can consider that state-of-the-art testing frameworks can address the challenges of assessing the quality of these apps. We report on the possibilities that can be considered with the current solutions.

Mocking the Google Nearby API. One of the solutions that a developer can consider consists in mocking the Nearby framework API. As a matter of example, we share a template file to illustrate how these apps can be unit tested using a mocked version of the Google *Nearby Connections API*.[8]

When mocked, with a framework like Mockito,[9] the Google Nearby Connections API hardens and checks the behavior of the app regarding the expected sequence of calls to the API. For example, if the app calls `acceptConnection` before calling `startDiscovery`, the test should fail. However, this approach remains limited to test the nearby apps as:

1) The peer interactions requires to be hardcoded, and can hardly cover all the possible scenarios,
2) The forged mocks may not really reflect the actual behavior of the underlying hardware components,
3) While the developer can test the app logic, she cannot perform black-box testing, which is required when testing context-based apps.

Testing with Physical Devices. Because it is hard to mock the Nearby hardware components, developers tend to consider physical devices to perform black-box testing using frameworks, like calaba.sh[10] or Cucumber.[11] However, this solution fails to scale testing with more than two devices, which quickly become a tedious task to automate and consider for any developer:

1) Only the simplest scenarios can be covered. It is not possible to automate black-box tests for outlier scenarios (*e.g.*, getting out of the connectivity range requires taking devices away from each other),
2) This approach fails to scale when considering deployment to hundreds of devices,
3) The test must be kept agnostic from the underlying hardware and the OS diversity,
4) When combining with cloud providers, the tests need to control the proximity between devices, and this proximity needs to be handled by the test.

Crowdsourced Testing with Beta-testers. One of the alternative solutions to test nearby mobile apps consists in using crowdsourced testing platforms to try the app on different mobile devices and manually follow the test scenarios [1]. Nonetheless, crowdsourced testing has some major drawbacks:

1) The tests are not reproducible,
2) The tests are not automated,
3) Covering all possible scenarios may take a long time.

Testing using simulators. Simulating the peers' behavior is possible to assess the connection behavior and peers interactions [2], [3]. This approach can help developers to test the interaction between peers, but it fails to test the exact behavior of the app, as mobile apps have their own dynamic context that cannot be simulated. Moreover, black box testing is not available in simulators, as there is no real app executed by the peer.

B. Objectives when Testing Nearby Mobile Apps

Unit testing is known to be a partial solution to assess the expected behavior of the app. In particular, when the execution context is known to be unpredictable, black-box testing becomes critical, especially for mobile devices [4]. Moreover, testing exhaustively a mobile app with every possible scenarios is desirable to reduce the risk of receiving bad user reviews and ratings on the app store. Beyond issues faced by standard mobile apps, the inclusion of the Google Nearby framework within a mobile app requires to address the following additional testing concerns:

[7] https://developers.google.com/nearby/connections/v11-update
[8] https://github.com/m3ftah/NearbyTest
[9] https://github.com/mockito/mockito
[10] https://calaba.sh/
[11] https://cucumber.io/

Peer Scalability. Nearby mobile apps are meant to support large number of users. Thus, developers are expected to test that their apps can effectively discover and potentially interact with a large number of nearby users;

Connection Resilience. Given the pervasive nature of the underlying network layer, the developer has to consider and test that unexpected disconnections of peers does not crash or negatively impact the nearby app;

Protocol Robustness. As nearby apps are interacting and exchanging messages by implementing an dedicated application protocol, the developer has to assess that the contextual interactions of peers does not lead to deadlocks;

App Interoperability. Given the device and OS fragmentation of Android,[12] different configurations need to be supported by the peers. More specifically, the nearby app is expected to work correctly, no matter the underlying configuration of nearby peers. Moreover, as Google Nearby also targets iOS, this implies that the test scenarios should be described in platform-neutral languages to maximize the interoperability of nearby apps;

App Reactivity. Nearby apps are expected to be opportunistic: by triggering some actions upon the discovery of a peer. This includes message broadcasting, forwarding a message to a specific user or detecting the maximum number of users. This reactive behaviour heavily depends on the configuration of the discovery protocol (*e.g.*, advertisement periodicity) to adjust the effectiveness of the triggered actions. Furthermore, these parameters might also have some critical implications for the end users (*e.g.*, battery consumption), thus forcing the developer to carefully tune them.

C. Challenges for Testing Nearby Apps

Given the elicitation of specific testing concerns and the limitation of state of practice when it comes to testing nearby mobile apps, we identified the following key challenges:

Emulators Support. Mobile apps that are using the Nearby framework can only be tested on physical devices and cannot be tested on emulators, or in the cloud (as Nearby requires some physical proximity between the tested devices). Even if the developer needs to test other functionalities (other than nearby features) using automated black-box testing on emulators, these tests will fail to execute without the hardware components associated to the Nearby framework;

Automation. The existing automation testing tools do not support testing nearby P2P mobile apps, so developers are expected to test their mobile apps manually, which is generally acknowledged as a poor approach for context-based mobile apps;

Scalability. Exploring the scalability of nearby mobile apps remains difficult. When interacting with a large number of nearby devices, the mobile app, including its user interface, has to adjust itself in order to visualize a large number of neighbors and the app needs to support a large number of communications without crashing or being forced to sleep by the mobile battery/resource manager;

Context Predictability. Nearby communications adopt an opportunistic communication scheme, which fits with the dynamics of the surrounding environment and the unpredictable nature of mobile user context. However, developers cannot anticipate and manually enumerate all the possible scenarios and test them to maximize the test coverage criteria. Furthermore, the continuous change in context information tend to explode the permutations of the available test space [4];

Cross-platform Tests. Both industrial and research communities did not provide any standardized way of testing nearby P2P mobile applications. Because the nearby framework needs to provide cross-platform communications, the test framework needs to be cross-platform too;

Black-box Testing. The interactions between nearby devices cannot be only covered by unit tests. The dynamic context events cannot be expected in controlled experiences and the nearby communication hardware cannot be statically mocked. Beyond the need to support integration testing scenarios, nearby apps also need to be tested in black-box due to the dynamic context, the variability of devices, APIs and screen sizes.

These challenges therefore pave the way of an extended support for nearby app testing, which we propose to cover with PEERFLEET, a scalable testing framework for nearby apps.

IV. THE PEERFLEET TESTING FRAMEWORK

The PEERFLEET testing frameworks leverages not only the challenges we identified, but also users' feedback from the Google Play store (cf. Table I) and the GitHub issues of the Google Nearby framework [5] to propose appropriate abstractions and primitives for testing nearby apps at large.

A. Framework Overview

The PEERFLEET testing framework intends to support the orchestration of a fleet of mobile devices, which can be physical devices connected to the development environment or virtual ones (*e.g.*, emulators), as depicted in Figure 1. PEERFLEET instruments these environments with a nearby adapter that connects a fleet of remote devices through a socket.io event bus. PEERFLEET exposes an API that is used to send specific commands to remote devices (cf. Listing 1). This API is used by the PEERFLEET orchestrator according to actions that are triggered by the test scenarios or evolutions in the devices vicinity. These evolutions are driven by the configured proximity dataset and the discovery protocol settings. The PEERFLEET API accommodates a wide diversity of black-box testing frameworks, such as monkeyrunner, Calaba.sh[13] or JBehave.[14] In particular, we provide a support for the

[12]https://developer.android.com/about/dashboards/

[13]https://calaba.sh

[14]https://jbehave.org

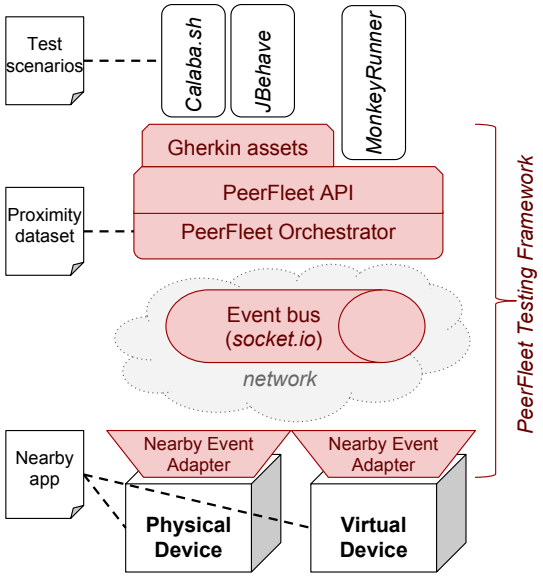


Fig. 1. Overview of PEERFLEET.

Gherkin specification language to leverage behavior-driven specifications.[15]

Listing 1. PEERFLEET API

```
public interface NearbyActions {
  void peerJoin(String peerId);
  void peerLeft(String peerId);
  void peerRequestConnection(String peerId);
  void peerAcceptConnection(String peerId);
  void peerRejectConnection(String peerId);
  void peerConnect(String peerId);
  void peerSendBytes(String peerId, byte[] bytes);
  void peerSendFile(String peerId, byte[] file);
  void peerSendStream(String peerId, byte[] stream);
  void peerDisconnect(String peerId);
  void nearbyPeers(String[] peers);
}
```

B. *Implementing the* PEERFLEET *Orchestrator*

The PEERFLEET orchestrator is composed of 2 parts : the PEERFLEET Test Runner and the PEERFLEET Bridge Component.

The PEERFLEET Test Runner is in charge of interpreting a testing scenario and interacts with the set of remote devices to check that the mobile apps behave accordingly to a given specification. For example, Listing 2 provides the snippet of the specification, based on Gherkin, of a nearby app that support device-to-device file transfer. This acceptance test describes 3 scenarios that can be triggered by the test runner according to the context. Whenever PEERFLEET detects two or more devices as nearby, it picks one of these three scenarios and executes it. The test runner completes when the bridge component notifies the completion of the proximity dataset.

The PEERFLEET Bridge Component is in charge of managing the interactions among the remote devices by controlling their Nearby framework through socket.io messages. Table III summarizes the mapping of the PEERFLEET framework API to socket.io events, which are produced by the bridge component and consumed by the nearby adapters of remote devices, depending on the call that are triggered from the PEERFLEET API.

[15]https://docs.cucumber.io/gherkin

Listing 2. Gherkin specification for a file transfer app based on Nearby

```
Feature: File exchange

  Rule: File transfer should never crash

    Background:
      Given there are <d> devices
      And discovery advertises every <x> seconds

    Scenario: File transfer should succeed
      Given there are more than one devices alive
      When 2 devices meet, they will connect
      And one device send a file
      Then the notification should be "file
transferred" on the screen

    Scenario: File transfer fails
      Given there are more than one devices alive
      When 2 devices meet, they will connect
      And one device send a file
      And one device disconnect
      Then the notification should be "peer
disconnected" on the screen

    Scenario: File transfer aborts
      Given there are more than one devices alive
      When 2 devices meet, they will connect
      And one device send a file
      And one device abort
      Then the notification should be "transfer
aborted" on the screen
```

TABLE III
MAPPING PEERFLEET API TO SOCKET.IO EVENTS

PEERFLEET API	Socket.io events
`peerJoin(id)`	`emit("peer_join",id)`
`peerLeft(id)`	`emit("peer_left")`
`peerRequestConnection(id)`	`emit("request",id)`
`peerAcceptConnection(id)`	`emit("accept",id)`
`peerRejectConnection(id)`	`emit("reject",id)`
`peerConnect(id)`	`emit("connect", id,p)`
`peerSendBytes(id,p)`	`emit("bytes",id,p)`
`peerSendFile(id,f)`	`emit("file",id,f)`
`peerSendStream(id,s)`	`emit("stream",id,s)`
`peerDisconnect(id)`	`emit("disconnect",id)`
`nearbyPeers(ids)`	`emit("nearby",ids)`

Figure 2 recapitulates the involved parties and interactions for when triggering the scenario `File transfer fails` introduced in Listing 2. The execution of this scenario can be controlled, step-by-step, from the testing environment (*e.g.*, developer laptop, continuous integration). The test runner initiates the bridge component by configuring the proximity dataset to be used, as well as the discovery settings for the nearby framework. This separation of concerns allows the developers to test the same scenario in different conditions and scales (number, density and diversity of devices). It also supports the investigation of the effects of the discovery protocol settings on the performance of the mobile app under test.

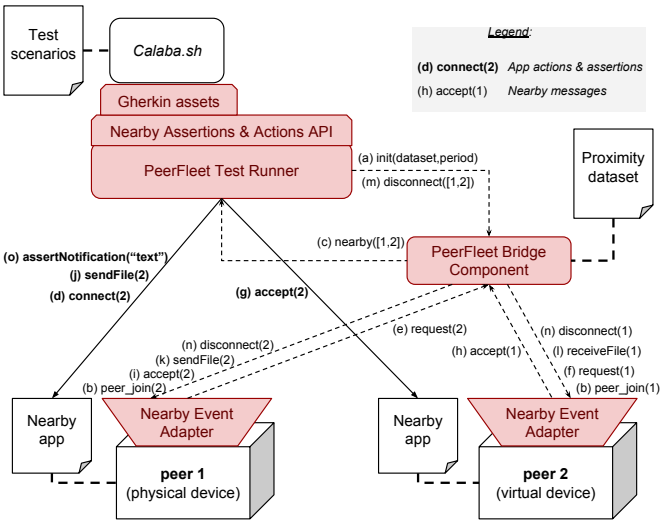


Fig. 2. Interactions among the PEERFLEET components.

Once the bridge component initialization process is completed, the test runner waits for the proximity notifications of the devices. The bridge component autonomously process the proximity dataset and, for each input step, it computes the potential proximity among devices, according to the discovery protocol settings. Upon detecting such a proximity, the bridge component fires a nearby event that triggers a scenario from the test runner. The execution of this scenario results in interactions with the instances of the mobile apps deployed in the virtual or physical devices. Each interaction can eventually result in calls to the Google Nearby framework, which are converted into socket.io events consumed by the bridge component and forwarded to the appropriate peers according to the current context state. Each peer can eventually react accordingly to the consumed events by producing an event that is further propagated by the bridge component.

V. EVALUATION

To evaluate the PEERFLEET testing framework, we focus on the following research questions:

RQ1: Does the PEERFLEET testing framework leverage the identification of bugs related to the usage of the Google Nearby framework?

RQ2: Can the PEERFLEET testing framework support the developer in setting the discovery protocol settings optimally

A. Experimental Setup for RQ1

To evaluate **RQ1**, we selected 8 Android open source apps (cf. Table V) to be tested with our PEERFLEET testing framework, we try to detect nearby bugs in these apps and, when no bug were detected, we artificially injected them to evaluate the precision of the bug detection. These mobile apps are run in the emulators, while the test runner executes a monkey that randomly chooses an action from the app UI and trigger the widget corresponding to it.

Classification of Nearby Bugs: In this paper, we consider the following classification of nearby bugs:

1) *Permission bugs* occur when the runtime permissions are revoked using the nearby framework syntax, which requires the app to request for these permissions;
2) *Scalability bugs* occur when the nearby app fails to discover and communicate with a large number of neighboring peers for several days, resulting in app crashes or if the emulator is out of memory;
3) *Protocol bugs* occur when the nearby app fails to establish the connection with remote peers in some contexts, because the nearby protocol is incorrectly implemented by the nearby app. Given that the Google Nearby framework specifies 17 error codes,[16] the mobile app under test may provide a partial implementation resulting in crashes or unexpected behaviors;
4) *Pervasive bugs* may occur when devices succeed to connect and, given the pervasive nature of their interactions, loose the connection, but the app is still sending data.

B. Experimental Setup for RQ2

Then, to address **RQ2**, we consider the following objectives:

- *To explore the maximum number of users.* This applies, for example, when considering a Bluetooth beacon in a store, which interacts with users passing nearby. As part of our evaluation, we used a modified version of the open source app HotMessages;[17]
- *To reach the maximum number of users.* For example, we consider the case of an emergency Android app that dispatches a message to the maximum number of users through Nearby framework. As part of our evaluation, we used a modified version of the open source app P2PMessaging. [18]

We believe that these two objectives cannot be only assessed by a simple guess, one developer rather needs to experiment with their mobile app on a representative dataset of users to estimate how much the objectives are reached. Then, developers can investigate which discovery protocol settings are the most suitable for their scenarios and objectives. Along with these objectives, the impact on the battery consumption can also be considered as an optimization objective.

Experiment Settings. To optimize the scalability or the performance of the objectives of her app, the developer can explore several discovery protocol settings:

1) *Discovery period*, which is the period between two consecutive discovering operations, each discovering operation takes 2 minutes. In our experiment, we have considered 3 discovery periods: 5 minutes, 10 minutes and 20 minutes;
2) *Connection strategy*, which is the topology used to connect the devices: *one-to-many* (WiFi) or *many-to-many* (Bluetooth).

[16]https://developers.google.com/android/reference/com/google/android/gms/nearby/connection/ConnectionsStatusCodes
[17]https://github.com/bigabig/hotmessages
[18]https://github.com/MrPKGupta/P2PMessaging

TABLE IV
CHARACTERISTICS OF DIFFERENT HAGGLE ONE PROXIMITY DATASETS

dataset	emulators (#)	duration (days)	interactions (#)	type
Intel	9	4.15	2,728	*mobile and stationary nodes*
Computer-lab	12	5.27	8,456	*mobile nodes*
Infocom2005	41	2.94	44,918	*mobile nodes*
Cambridge-city-complete	52	11.42	21,746	*mobile and stationary nodes*
Infocom2006-short-range	78	3.87	257,958	*short-range nodes*
Infocom2006-complete	98	3.90	341,202	*short-range and long range nodes*

To answer **RQ2**, we therefore adopt the following experiment protocol:

1) We launch a crowd of Android mobile emulators (composed of 9, 12, 41, 52, 78 and 98 devices), depending on the size of the associated proximity dataset [6]—the characteristics of each dataset are reported in Table IV;
2) The PEERFLEET test runner executes an acceptance test that randomly chooses an action from the app UI and trigger the widget corresponding to it;
3) These devices will interact with each other following the *Bluetooth* or *Wifi* encounters in both *indoor* or *outdoor* situations depending on the proximity dataset;
4) For each run, the new variant explores a different combination of settings for discovery period and connection strategy;
5) At runtime, we collect statistics for each objective from the bridge component.

C. Proximity Datasets

The datasets used in our experiments are based on The Cambridge/haggle dataset (v. 2016-08-28) available from the CRAWDAD database [6], which contains Bluetooth proximity traces for a number of days with indoor and outdoor environments, the characteristics of the used datasets are summarized in Table IV.

D. Battery Consumption

In order to show the effect of each discovery period on the battery consumption, we deploy mobile apps with PEERFLEET on two physical devices: Moto Z (Android 7.1.1) and Galaxy Tab A (2016, Android 5.1.1). We have run the HotMessages app on the two phones, with different discovery settings per run. For each variant, we keep the phone screen unlocked to avoid the app being sent to the background and to disable the battery manager strategies, which can put the app on idle state and interfere with the measurements. The measurements were collected using the `adb shell dumpsys batterystats` command that provides statistical data about battery usage on the device.

VI. EXPERIMENTAL RESULTS

A. RQ1: Does the PEERFLEET testing framework leverage the identification of nearby bugs?

The results for the nearby apps under test are reported in the table V. The apps were tested against the classes of nearby bugs identified in Section V-A. Table V reports if, for each app under test, PEERFLEET succeeds to identify an occurrence of nearby bug.

Discussion. One can observe that each vanilla app under test exhibited at least one class of nearby bug that could be detected by PEERFLEET testing framework. Interestingly, PEERFLEET succeeds to detect *protocol bugs* in all the mobile apps we selected, which highlight the partial implementation of the nearby support, thus failing to work in a wide diversity of execution conditions. Typically, after manual investigation, we could observe that most of the mobile apps manage only one of the 17 error codes (`STATUS_OK`), thus leading to app crashes when it faces unexpected situations. Regarding *permission bugs*, most of the mobile apps seem to correctly support the runtime permissions. We believe that the low rate of detected bugs, is related to the associated Android linter rules, which warn the developers about this error. Yet, one can observe that PEERFLEET succeeds to detect the artificial bugs we injected. Exploring *scalability bugs* requires to stress the mobile app by considering more complex proximity datasets that exhibit a large density of devices and potentially long periods of executions. In such situations, PEERFLEET highlights symptoms such as the app UI fails to display multiple peers, the app fails to accept connections from more than one peer, and the app fails to properly disconnect from peers—*i.e.*, resources are not properly released and the app cannot reconnect to the same peer. Finally, regarding the occurrences of *pervasive bugs*, we observe that the developers of the incriminated mobile apps missed to properly handle connection lost when implementing the listener `EndpointDiscoveryCallback.onEndpointLost()`, which is triggered whenever a peer disappears from the device's vicinity.

B. RQ2: Can the PEERFLEET testing framework support the developer in setting the discovery protocol settings optimally?

Applications' performances may depend on a large diversity of parameters, ranging from network interface efficiency, to hardware capabilities, to code optimizations. In our study, we are interested in code-level optimizations that positively impact the app performances.

In particular, Tables VI and VII reports on the collected statistics of the experiment we described in Section V-B.

Table VI plots the percentage of missed peers (nearby dataset peers failed to be detected by the mobile app) depending on the discovery period. As expected, the shorter the period is, the less number of peers are missed. However, PEERFLEET

TABLE V
OPEN SOURCE ANDROID NEARBY APPS COVERAGE

Apps — Bugs	Permission	Scalability	Protocol	Pervasive
com.p2psample	Injected	Found	Found	Injected
com.example.nearbyplayground	Injected	Injected	Found	Injected
de.bigabig.hotmessages	Injected	Found	Found	Injected
com.example.hotelca.poolnumbergenerator	Injected	Injected	Found	Found
com.jhony.jester.play	Found	Injected	Found	Found
com.google.location.nearby.apps.walkietalkie	Injected	Injected	Found	Found
com.google.location.nearby.apps.rockpaperscissors	Injected	Injected	Found	Found
com.supercilex.autohotspot	Injected	Injected	Found	Found

TABLE VI
EXPERIMENT STATISTICS - EVOLUTION OF MISSED PEERS FOR THE HOTMESSAGES APP

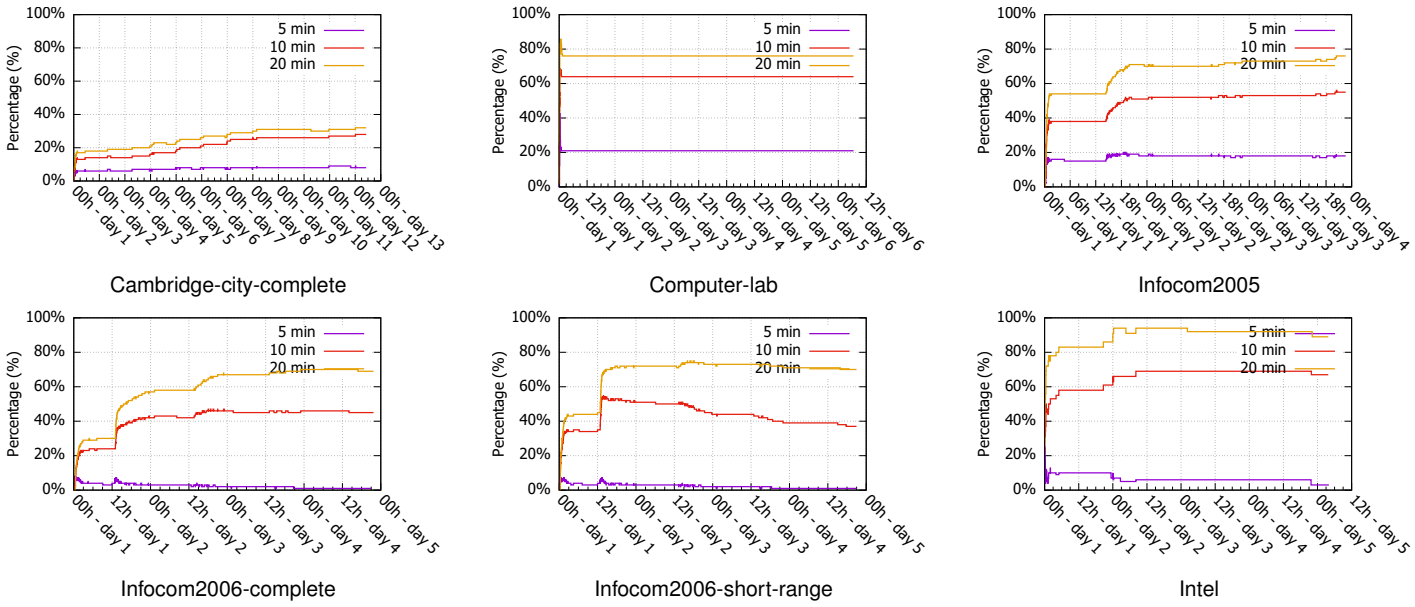
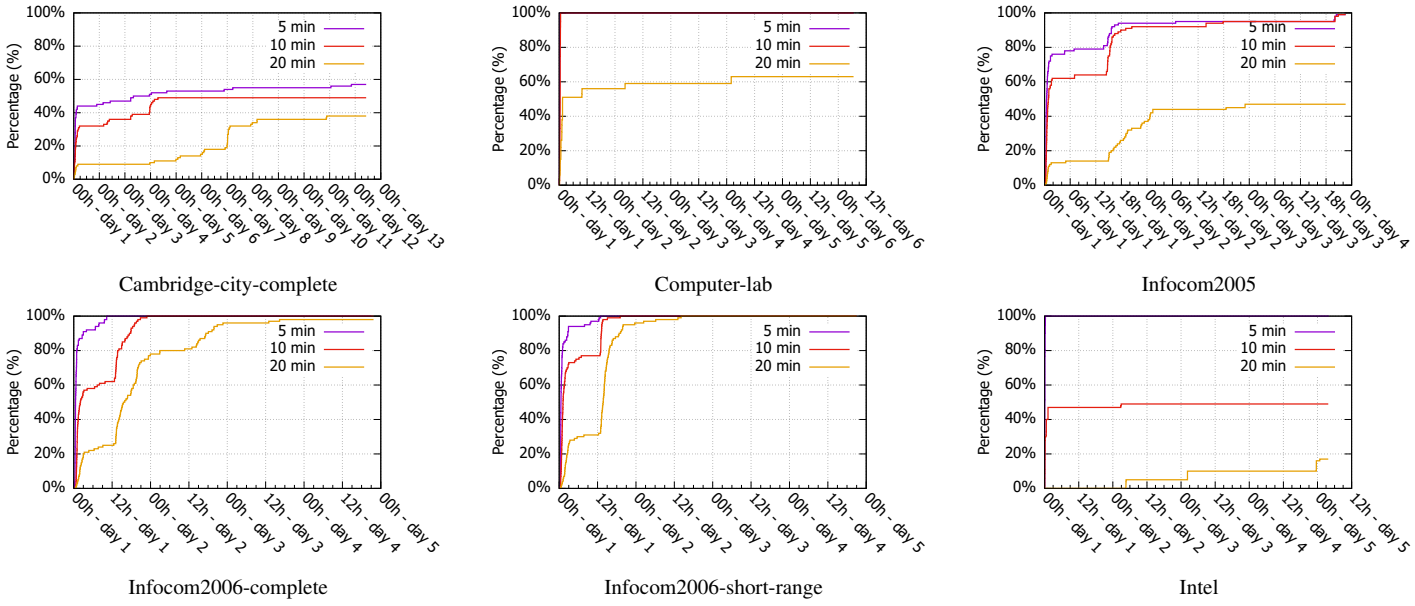


TABLE VII
EXPERIMENT STATISTICS - EVOLUTION OF REACHED PEERS FOR THE P2PMESSAGING APP



succeeds to deliver some concrete insights on the device coverage in the emulated conditions depending on the settings values. Thus, the developer can take informed decisions on the most appropriate values thanks to PEERFLEET.

Similarly, Table VII reports on the percentage of reached peers. Again, the $5min$ discovery period setting maximize the number of reached nodes. More surprisingly, one can observe that the $10min$ discovery period results performs close to the $5min$ settings (except for one dataset) meaning that the $5min$ discovery period is not really needed especially if the app is intended to run for several days (the two lines converge). We can also note that the number of reached nodes depends also on the number of the dataset nodes, so if the developer expects to have more than 50 colocated peers, then choosing the $20min$ discovery period will get the same results as the $5min$ discovery period. If the developer estimates that the number of users is less than 10 (like in the *Intel* dataset), then choosing $20min$ will lead to poor performances, thus recommending the developer to set $5min$ as discovery period.

Table VIII reports on the percentage comparison between missed and reached peers using Bluetooth and WiFi connections. We can see that there is no big difference in the percentage of missed peers, but there is a big difference in the percentage of reached ones. This is due to the simultaneous connections that the Bluetooth connection offers over the WiFi connection. While apps using WiFi connection (*one-to-many*) cannot initiate another connection during the current session, the apps using Bluetooth connections (*many-to-many*) can pair with several devices without the need to disconnect any one of them. Thus, adopting the Bluetooth connection strategy can better support the efficient dissemination of messages compared to the WiFi strategy.

Finally, Table IX reports on the power measurements obtained from physical devices connected to PEERFLEET. The *witness* app setting disables the discovery protocol. We can see that different discovery periods have a direct impact on the battery consumption that can be noticed by the user. For example, a Moto Z user who installs the app configured with `5 min` discovery period, may notice that the app is responsible for consuming 52% of the battery power, thus encouraging her to uninstall it.

Discussion. Depending on the objectives and the number of the targeted users, the developers can tune their app settings. For example, if the main objective is to disseminate a message to the maximum number of peers, the setting `20 min` discovery period will give good results (up to 98 %).

When deciding between using Bluetooth or WiFi. If the developer's objective is to exchange messages between direct users, given that WiFi and Bluetooth have almost the same missed peers percentage, the decision can depend on other factors, like : the active connection method or the other peer's chosen connection method. If the developer objective is to disseminate the message to maximum number of users than Bluetooth is the best fit.

By running these experiments, developers and product managers can gather some insights on how to choose the best parameters and connection strategy, depending on the user context. Thanks to Google Nearby, the developer can also consider the implementation of an hybrid strategy, thus combining Bluetooth and WiFi, to detect neighbors using Bluetooth then exchange data using WiFi to benefit from the high bandwidth transfer of the WiFi connection.

The PEERFLEET testing framework therefore supports the exploration of app performance along different dimensions, including the proximity dataset characteristic but also the device fragmentation and discovery protocol settings. This exploration of the app performance space can guide the developer in tuning these parameters optimally according to her requirements.

C. Threats to Validity

While the goal of our study is to increase the quality of mobile apps by detecting the nearby bugs through the application of black-box testing at large, a threat lies in the possibility that we missed some classes of nearby bugs.

Another threat lies in the implementation of our PEERFLEET testing framework. In particular, the nearby event adapter provides an emulated support of the hardware discovery mechanism that is implemented by the operating system. Although we extensively tested our implementation of this nearby event adapter, we cannot be sure that it faithfully complies with all the implementation of the Google Nearby framework, and their potential bugs.

Finally, a possible threat lies in our experimental framework. We did extensive testing of our experimentation, and we manually verified the data from our experiments. However, as for any experimental infrastructure, there may be bugs. We hope that they only change marginal quantitative results and not the quality of our contribution.

VII. RELATED WORK

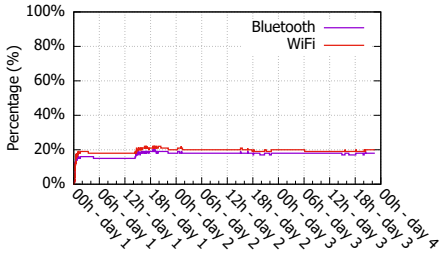
Testing mobile apps has taken attention in recent years both in industrial and research due to the proliferation of mobile devices and the need to deliver quality apps to demanding mobile users.

In our framework we propose new test oracles, some literature work also propose to add a set of test oracles [7], [8], [9]. We test nearby P2P mobile apps which lead us to consider the dynamic context of the app being tested [10], [11], [12], [13], [4], and to consider controlling the executed event sequences in the tested app [14], [15], [16], [17]. Also, in our framework, we propose a black-box testing as it is becoming a requirement for mobile apps testing [18], [19], [20].

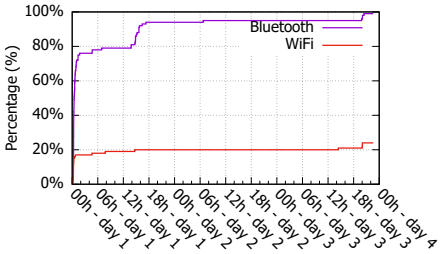
While our approach is based on test automation [21], [22], [23], we do not propose any new test automation framework. Instead, we build upon existing test automation frameworks. Linares-Vasquez *et al.* [24] overview the state of the art of the test automation solutions for mobile apps.

Meftah *et al.* [25], propose a tool to perform black box testing for WiFi P2P Android apps, similar to us they mock the hardware component, but we propose a more generic framework for all nearby P2P connection APIs, with a rich

TABLE VIII
EXPERIMENT STATISTICS - IMPACT OF THE CONNECTION STRATEGY (USING INFOCOM2005)



Missed peers



Reached peers

TABLE IX
IMPACT OF THE DISCOVERY PROTOCOL SETTINGS ON BATTERY LIFESPAN

Settings	Moto Z	Galaxy tab
witness	1d 04h 46mn 26s	1d 03h 38mn 40s
`20 min`	17h 04mn 44s	1d 01h 25mn 55s
`10 min`	16h 44mn 42s	1d 01h 02mn 49s
`5 min`	13h 36mn 26s	23h 45mn 38s

set of assertions and actions. While they propose some low level test assertion, we propose high-level test oracles to test the interaction between peers using the framework assertions.

Zhang *et al.* [1] present a crowd-sourced testing approach to better test real-life scenarios with freelancer testers. While this approach can be used to test Nearby P2P mobile apps, the tests are not reproducible and cannot be automated. Also, it will take the test a lot of time to cover all the possible scenarios. This approach can be used along with our testing approach, but it cannot replace it.

Serfass *et al.* [3] propose a simulation solution for Android apps that are using NFC P2P data exchange. Similar to us, they propose a testing solution for apps interacting with the hardware component. Their solution is based on simulation, but our solution proposes to run the app on peers to get real interactions between peers and test these interactions.

Wen *et al.* [26] present a framework for a parallel UI testing for Android apps. While their intention is to speed up the tests, our intention of using parallel testing is to run the app on different devices that interact with each other in a synchronized way.

To the best of our knowledge, we are the first to explore and propose a testing solution for nearby P2P mobile apps.

VIII. CONCLUSION

Developers are facing problems developing P2P Nearby apps, there are no guidelines for testing and developing such apps. In this paper, we proposed a test framework called PEERFLEET, this framework will help developers debug and test their P2P Nearby apps, the framework will also help the developers find the best set of parameters that are the most suitable to their objectives.

Further research routes would be extending this testing approach and generalizing it for all mobile P2P communications, and adding custom support for wireless printers, wireless headphones and car stereo. There is also a need to detect further scaling issues and document them as guidelines for developers.

REFERENCES

[1] T. Zhang, J. Gao, and J. Cheng, "Crowdsourced testing services for mobile apps," in *Service-Oriented System Engineering (SOSE), 2017 IEEE Symposium on*. IEEE, 2017, pp. 75–80.

[2] B. Richerzhagen, D. Stingl, J. Rückert, and R. Steinmetz, "Simonstrator: Simulation and Prototyping Platform for Distributed Mobile Applications," *EAI International Conference on Simulation Tools and Techniques (SIMUTOOLS)*, pp. 1–6, 2015.

[3] D. Serfass and K. Yoshigoe, "Wireless sensor networks using android virtual devices and near field communication peer-to-peer emulation," in *Conference Proceedings - IEEE SOUTHEASTCON*. IEEE, apr 2013, pp. 1–6.

[4] I. d. S. Santos, R. M. d. C. Andrade, L. S. Rocha, S. Matalonga, K. M. de Oliveira, and G. H. Travassos, "Test case design for context-aware applications: Are we there yet?" *Information and Software Technology*, vol. 88, pp. 1–16, aug 2017.

[5] G. Developers, "Issues," 2017. [Online]. Available: https://github.com/googlesamples/android-nearby/issues

[6] D.-G. Akestoridis, "CRAWDAD dataset uoi/haggle (v. 2016-08-28): derived from cambridge/haggle (v. 2009-05-29)," Downloaded from https://crawdad.org/uoi/haggle/20160828/one, Mar. 2018, traceset: one.

[7] Y. D. Lin, J. F. Rojas, E. T. H. Chu, and Y. C. Lai, "On the accuracy, efficiency, and reusability of automated test oracles for android devices," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 957–970, oct 2014.

[8] M. Fazzini, E. N. D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests," in *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*. IEEE, mar 2017, pp. 149–160.

[9] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated test oracles for GUIs," *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 6, pp. 30–39, 2000.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, and N. Amatucci, "Considering context events in event-based testing of mobile applications," in *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2013*. IEEE, mar 2013, pp. 126–133.

[11] T. A. Majchrzak and M. Schulte, "Context-Dependent Testing of Applications for Mobile Devices," *Open Journal of Web Technologies*, vol. 2, no. 1, pp. 27–39, 2015.

[12] O. E. K. Aktouf, T. Zhang, J. Gao, and T. Uehara, "Testing location-based function services for mobile applications," in *Proceedings - 9th IEEE International Symposium on Service-Oriented System Engineering, IEEE SOSE 2015*, vol. 30. IEEE, mar 2015, pp. 308–314.

[13] W. Song, X. Qian, and J. Huang, "EHBDroid: Beyond GUI testing for Android applications," in *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, oct 2017, pp. 27–37.

[14] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, p. 67, 2013.

[15] H. Zhu, X. Ye, X. Zhang, and K. Shen, "A Context-Aware Approach for Dynamic GUI Testing of Android Applications," in *Proceedings - International Computer Software and Applications Conference*, vol. 2. IEEE, jul 2015, pp. 248–253.

[16] S. Yu and S. Takada, "Mobile application test case generation focusing on external events," in *Proceedings of the 1st International Workshop on Mobile Development - Mobile! 2016*. New York, New York, USA: ACM Press, 2016, pp. 41–42.

[17] C. Quist and A. Møller, "Systematic Execution of Android Test Suites in Adverse Conditions," *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, pp. 83–93, 2015.

[18] B. Jiang, X. Long, and X. Gao, "MobileTest: A tool supporting automatic black box test for software on smart mobile devices," in *Proceedings - International Conference on Software Engineering*. IEEE, may 2007, pp. 8–8.

[19] K. Mao, M. Harman, and Y. Jia, "Robotic Testing of Mobile Apps for Truly Black-Box Automation," *IEEE Software*, vol. 34, no. 2, pp. 11–16, mar 2017.

[20] T. Yumoto, T. Matsuodani, and K. Tsuda, "A test analysis method for black box testing using AUT and fault knowledge," in *Procedia Computer Science*, vol. 22, 2013, pp. 551–560.

[21] K. Mao, M. Harman, and Y. Jia, "Crowd intelligence enhances automated mobile testing," in *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, oct 2017, pp. 16–26.

[22] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 559–570.

[23] L. L. Zhang, C. J. M. Liang, Y. Liu, and E. Chen, "Systematically testing background services of mobile apps," in *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, oct 2017, pp. 4–15.

[24] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing," *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*, pp. 399–410, 2017.

[25] L. Meftah, M. Gomez, R. Rouvoy, and I. Chrisment, "ANDROFLEET: Testing WiFi Peer-to-Peer Mobile Apps in the Large," *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 961–966, 2017.

[26] H. L. Wen, C. H. Lin, T. H. Hsieh, and C. Z. Yang, "PATS: A Parallel GUI Testing Framework for Android Applications," in *Proceedings - International Computer Software and Applications Conference*, vol. 2. IEEE, jul 2015, pp. 210–215.