Performance Assessment of mHealth Apps

Gerson Rodriguez[1], Devasena Inupakutika[1], Sahak Kaghyan[1], David Akopian[1], Palden Lama[1], Patricia Chalela[2], and Amelie G. Ramirez[2]
[1]University of Texas at San Antonio, [2]University of Texas Health Science Center, San Antonio, TX, USA
Gerson.rodriguez.gr@gmail.com, mmn609@my.utsa.edu, sahak.kaghyan@gmail.com, david.akopian@utsa.edu, palden.lama@utsa.edu, chalela@uthscsa.edu and ramirezag@uthscsa.edu

***Abstract*— Mobile Health (mHealth) apps are being widely used to monitor the health of patients with chronic medical conditions with the proliferation and the increasing use of smartphones. Mobile devices have limited computation power and energy supply which may lead to either delayed alarms, shorter battery life or excessive memory usage limiting their ability to execute resource-intensive functionality and inhibit proper medical monitoring. This paper presents a methodology for measurement-based performance assessment of cloud backend and mobile networks that support mHealth services. The methodology targets the assessment of a prototype mHealth app developed for breast cancer patients undergoing Endocrine Hormone Therapy (EHT). It models third-party cloud backend services to examine the performance in a representative testing scenario for end-users accessing the app. Experimental results are reported and compared for native Android and iOS implementations. The analysis further reflects the impact of the network and device battery conditions on response times and end-user quality of experience. The contribution of this work is twofold: (a) First, it presents a performance methodology and analysis of a fully functional medication adherence management mHealth app implemented on major duopoly of mobile platforms (android and iOS) and (b) Second, based on the performance analysis, conclusions are drawn that serve as the recommendation pathway for the development of similar medical reference mobile apps.**

Keywords—mobile applications, cloud services, performance, smartphones

I. Introduction

Mobile devices bring people closer and enable ubiquitous connection by providing responsive services with rich content, such as, messaging, push-notifications, high definition and live videos. Almost all the mobile apps that we use on a daily basis (e.g., Fitbit, Instagram, YouTube, Uber, and Google Maps) are dependent on external web services, network connectivity (e.g., Wi-Fi, Cellular networks (LTE), Bluetooth Low Energy (BLE)) and sensors that are continuously interacting. However, such rapid growth of apps, the richness in their features and the remarkably increased network traffics cause overhead to the device resources consumption such as power, CPUs, bandwidth etc. and the mobile network. These aspects attract attention from mobile apps and full stack researchers and developers for gaining a better knowhow of mobile apps' resources usage and their effects on the communication with a backend database. Context awareness (especially in the case of mHealth apps), mobile device fragmentation, event-driven programming paradigm (mostly triggering activities based on user-interaction in this case) and ever changing mobile technologies and frameworks considerable challenges for the software correctness and performance. Hence, considering the time and effort required for consolidating all the possible test scenarios for testing and generating the corresponding test cases, automated testing is crucial in performance assessment and delivery of good quality apps.

This paper focuses on analyzing the parameters affecting the mobile apps performance on Android and iOS platforms under varying load to a cloud-based backend database (DB) server. Major emphasis is put on the response time ($T_{response}$) as greater number of applications and services such as healthcare [2], file-hosting, mobile cloud sensing, location-based mobile services, mobile commerce, mobile learning etc. are delay sensitive. It is an important performance metric for user experience. Given that the choice of a testing framework depends on specific testing goals and research context, we devise a performance methodology extending the continuous, evolutionary, and large scale (CEL) [1] principles. We further study the performance of a case-study mobile app from the perspective of smartphone's memory and battery modes as resources usage under different mobile network connectivity (cellular vs Wi-Fi) and the response times of a service request from a real-time cloud DB with varying loads. Consequently, our work aims to provide a direction to mobile app testing by improving the model-based representations of the mobile apps and generating test cases specific to testing goals.

As a case-study medical reference and point-of-care app, we developed a cross-platform HT Patient Helper app which exploits rich existing resources and development tools from Google and Apple. For the evaluation purposes, the app includes integration of new generation of mobile and cloud computing and messaging solutions for healthcare. The app is designed to incorporate most of the aspects of the EHT for breast-cancer patients [3] that includes relevant disease, symptoms, medication, healthcare centers, doctor/ healthcare contact information along with the images and videos of symptoms management, and medical procedures from peers and doctors. The backend services provided by Google Firebase called Firebase Cloud Messaging (FCM) [4] are used to monitor and send messages and notifications to the user. Firebase is a Google platform that facilitates cloud storage and sends notifications to mobile devices through a cloud manager. Key features of the app include symptoms monitoring, educational content and videos, peer modelling closed group that helps patients learn about the symptoms and available resources, understand the intensity of the symptom, improve and gain coping skills to manage the level of tension and build a network of support to enervate the fear of recurrence. We conduct extensive performance analysis by observing various performance parameters specific to load on the real-time database, and the effect of load, device battery life and mobile network type on the app's response times. This analysis will also allow us to find the aspects in the app design that affect

The study is supported in part by Susan G. Komen (Award No. SAB160005), the Mays Cancer Center (Grant No. P30 CA054174) and Redes En Acción (Grant No. U54 CA153511).

overall performance which otherwise are not evident for end-users (patients).

The outline of the paper is as follows: Section II presents the related works on the use of cloud platforms that provide the DB as a service and introduces state-of-the-art studies towards performance analysis utilizing the cloud technology. Section III discusses our testing methodology applied in assessing the performance through our developed prototype mHealth HT Patient Helper app and summarizes the dataset for the user scenario. Section IV presents experimental results for the tests conducted with different battery and network modes. Finally, conclusion remarks are made and the future work is discussed in Section V.

II. Related Work

A. Cloud Database As-a-Service

Cloud computing is emerging as an alternative for researchers and practitioners to clusters, grids and production environments. In the current work, we are interested in the cloud DB as a service paradigm [5] that can support numerous internet-based mobile apps. Due to the advantages of cloud computing for the deployment of data-intensive apps such as resources elasticity, pay-as-you-go cost model, scalability etc., it has become possible to achieve higher throughput by continuously adding computing resources such as DB servers if the workload increases. The authors in [6] provide a comprehensive overview of the different options of deploying the DB tier of software apps on cloud platforms. Consequently, the cloud is considered to be a good alternative that assures safety of the data and easily accessible on a need basis [7]. However, in terms of performance with respect to data retrieval and access, there are several concerns that have to be considered. There are several studies [8], [9], [10] that focus on comparative analysis between cloud and traditional DBs but from the architectural perspective. Latest work by the authors in [11] presented the analysis based on a quantitative approach for performance comparison and the analysis between on-premises and Azure SQL Server DBs. Similar to [11], we focus on $T_{response}$ of the cloud DB as a metric for the analysis. As stated in section I, we leverage Google Firebase as a cloud-hosted NoSQL DB for the current work. Firebase was the popular choice for the current case-study app as it offers convenient backend services (authentication, file storage, analytics, cloud messaging (FCM), real-time DB, cloud firestore and data synchronization), easy to use, fast publisher/ subscriber, and offers a quicker method to prototype similar mHealth-based apps. Additionally, we report the 95th-percentile response time of the test scenario requests. In the next subsection, we briefly cover the necessity of measurement-based performance testing for mHealth apps.

B. Mobile Apps Performance Challenges Effecting Healthcare

With the rapidly increasing mHealth apps, there is an increased need for the app development processes to include the assessment of the app's performance. There have been previous formative works as in [12] that report user-centered design processes for the app design in particular to mHealth apps, however they haven't touched on the cloud-based performance aspect. There are significant previous efforts by researchers [13], [14], [15] and mobile app designers in presenting the developers with the best practices and guidelines for improving the performance of the mobile apps, there is still a noticeable gap between research and practical deployment in terms of addressing performance bottlenecks by the developers. A review [16] of mHealth based biomedical apps focuses on the communication aspect (i.e. connectivity or link technology) between medical devices or users and the data collector (PN or backend database) in a mHealth system. The authors believe that the latency of such apps has to be limited to a range between a few seconds to 1 minute for seamless performance as the mHealth apps could be life dependent with the highest priority. [14] and [17] analyzed Android and iOS apps respectively and found that the unresponsiveness and significant resource usage are the major contributions for the negative reviews by the users. These factors are the prime concerns that cannot be compromised when it comes of mHealth apps that require users/ patients to keep track of their symptoms or reminders for medication.

In the current work, we analyze the performance of our HT Patient Helper app by identifying the factors that impact the app-cloud communication. In particular, we start with the initial versions (both Android and iOS) of our prototype HT Patient Helper app. We then analyze the impact of device's battery conditions (full battery and power saving mode) and network connectivity (LTE cellular network and 802.11 WiFi) with varying loads on *Firebase* on the response time.

III. Testing Methodologies

This section presents testing scenarios for the prototype HT Patient Helper Android and iOS apps which approach realistic and representative app usages to assess the performance. We conducted three assessment testing campaigns to observe the Firebase cloud performance and apps' capabilities for the test scenario: *Firebase loading* (emulating many devices to put stress), *Device battery* effect on the network and *Cellular (LTE) vs Wi-Fi* on the performance of the apps' connections with Firebase. For improved and seamless analysis for Firebase loading, initially, we identified performance bottlenecks in the app that are not evident to end-user but have effect on the overall performance. We provide a brief overview of app, test platform, UI Automation testing, test dataset and test scenario before the descriptions about the three testing campaigns.

A. Overview of the App

Fig. 1 shows the overall system architecture of the HT Patient Helper app with Firebase as the cloud DB server. User-specific information, symptoms, push notifications and videos reside in Firebase as corresponding data models. It provides 128-bit encryption on all datasets and secured user connections. It also makes the storage and client-server communication simpler. The educational content is deployed centrally using GitHub pages for medical educational management and to provide them access to broader information about the symptoms and HT itself. Both Android and iOS versions of the app use the same centralized Firebase DB nodes. The doctors, PN or admin are provided with a web-based interface to monitor the data received at the cloud DB. The app in general has the following features:

1. **Educational content:** Breast cancer specific symptoms, medication terminology and videos from doctors and peers.

2. **Symptoms tracking:** Enables patients undergoing therapy to keep track of their symptoms regularly, directs them to corresponding educational material as well as visualization of their progress in the form of symptom summary graphs.

3. **Notifications:** Receives push notifications for symptoms submission, reminders, doctor appointments and messages to keep them motivated throughout the intervention.

4. **Calendar:** Setting reminders for appointments and symptoms tracking.

5. Ability to contact PN and technical support team through phone calls, *SMS* or Email in case of emergency and technical difficulties respectively.

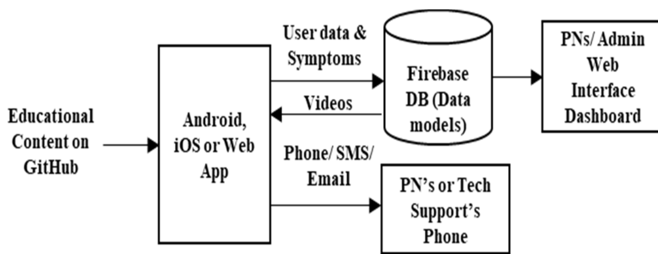

Fig. 1. System Architecture of HT Patient Helper App

Test Scenario: Accessing images and videos of medical procedures and diseases is one of the core features of medical reference apps. Hence, we model the test scenario that consists of the navigation sequence relevant to retrieval of breast-cancer videos on tips to manage symptoms from doctors and peers: *Splash page→Login page→Home page→HT Learn More (Educational content)→ Videos tab.*

B. Test Platform

The equipment used for conducting experiments are as follows:

1. *Test Computer:* 2015 Macbook Pro 15" running macOS Mojave, Processor 2.5 GHz i7 with 16GB 1600 MHz DDR3.

2. *Test Mobile (Android):* Nexus 6P Octa-core (4x1.55 GHz Cortex-A53 & 4x2.0 GHz Cortex-A57) running android 8.0 Oreo.

3. *Test Mobile (iOS):* iPhone XS Hexa-core (2x2.5 GHz Vortex + 4x1.6 GHz Tempest) running iOS 12.2.

C. UI Automation Testing

Automated testing using scripting techniques improves the effort per unit time and the accuracy per each test case. This section describes the Android and iOS based mobile UI testing performed as part of the performance analysis. UI testing can be performed in a variety of ways. The easiest option is to execute manual tests. Automated UI testing defines the sets of the scripts to be executed rigorously in a quick, repeatable way irrespective of the diverse functionalities of the app interface based on its features. UI automated tests are a better way to recognize the presence of regressions in the transition between consecutive releases of the app. It is also observed that in our current work, manual testing is problematic as it doesn't allow control over the delays between views and is not consistent between tests. We used UIAutomator [18] with Espresso [19] and XCTest [20] testing frameworks for testing and generating repeatable testing scripts on Android and iOS respectively. These frameworks are specifically used for simulating user interactions.

D. Test Dataset

In the current work, we use a massive video dataset to load the Firebase for the performance analysis. For the video dataset, we adopted publicly available videos from *Youtube.com* and other video sites. We used a command line tool *youtube-dl* [21] to download the list of video URLs or links. As discussed before, our video data resides on Firebase. We modeled video data to include them based on the sections. The *videos* node inside Firebase consists of *sections* and *videos* as nested nodes. *Section* node has section names as keys. To that end, each section name has associated one or more videos from *videos* node. Videos are identified by their section names. They are further modeled to consist of thumbnail image, URL and video title as key. After the retrieval of the massive videos dataset, we developed a *Python*-based generator to export the data according to the aforementioned video data model into a *json* file and further to our Firebase project. Hence, to load the Firebase, we retrofit the database by populating the *videos* node with more than 3000 sections and around 5000 videos.

E. Test Scenario in detail

In this section, we describe the test scenario for the simulations. UI model is an important element in model-based testing. A test case includes a complete path from starting initial to a final state in the test scenario. We implemented test cases using the UIAutomator (android) and XCTest (iOS) frameworks. For all the transitions, the event is translated to corresponding framework's API calls that trigger the event. Fig. 2 shows the UI view of the testing use case of educational videos retrieval from Firebase. It leverages dynamic loading that removes the need of re-publishing the app every time the data changes. On Firebase, the root node *videos* under the app root consists of *section*s and *videos* as child nodes that displays section titles and video thumbnails with title further.

F. Tests Design

This section describes our current testing scenario and the three tests conducted with that scenario. The scenario consists of users signing into the app, accessing HT Learn More educational video view followed by retrieving the sections.

1) Loading the Firebase

Since, the sections are retrieved from the Firebase, it is hypothesized that as the Firebase load increases, the response time should increase. Hence, we conduct three sets of experiments to observe the response time variations that are described next followed by Android and iOS comparisons. For our current version of apps, we use Firebase under the free subscription primarily because the app is currently at beta testing phase, and only 8 breast cancer patients and

professionals from healthcare research team are currently using it. The protocol for the intervention is described in [3]. This allows only 100 simultaneous users, 10 GB of bandwidth and 10 GB of storage. We started loading Firebase by querying the entire *videos* database at different time intervals. A separate Java application was created to control different threads to simulate a user and submit a request to Firebase. A bash test automation script was then used to control the spawns, iterations, simultaneous users and the requests to cloud DB. The load graph on the Firebase console shows how much the database is in use, processing the app requests, over a limited one-minute interval. The console also represents only the highest load in a given hour. Hence, if the target is 25%, and the test run displays 30%, one must wait an hour before running another test. To accelerate this process, we created three separate Firebase projects for the performance module of the app.

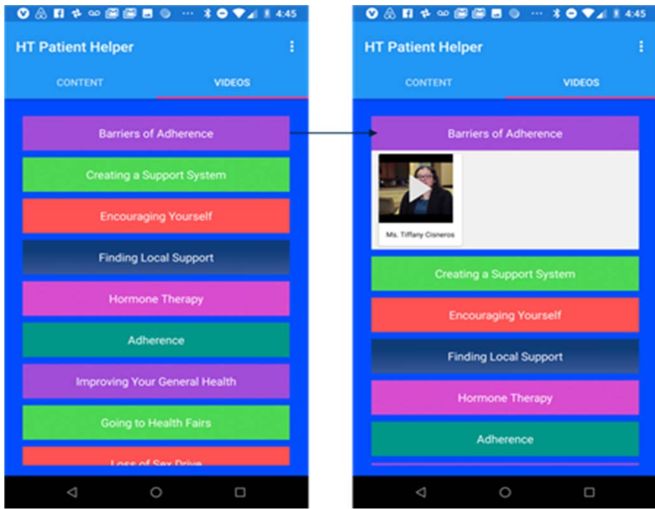

Fig. 2. Educational Videos UI View (Sections and Videos)

To have proper control over the delays between views and consistency between tests, performance analysis of app's section retrieval control flow from no load on Firebase to full load (0-100%) in increments of 25%, also models delay in LTE and Wi-Fi connections taking into consideration energy characteristics (device battery). Fig. 3. shows the events, states and the complete path further in detail for the test scenario in the current work. The simulations go through this path in every iteration.

2) Device Battery Test

In our current testing scenario of section retrieval for users or patients using the app, the system response time directly affects the user experience. The response is affected by the device resource utilization i.e. the energy consumption of different system components. Hence, device battery also contributes to response time. For this reason, we performed a test by changing the device battery mode to and from the *Regular* and *Power Saving (PS)* modes. We also analyzed the effect of device battery modes in the presence of varying load.

3) Cellular LTE vs Wi-Fi Test

This test assesses the impact of available mobile wireless networks connectivity on the app's interactions and response time for the testing scenario. This is again in the presence of dynamic Firebase loads. Typically, we conducted a study of these interactions and their impact on the 95th percentile of the response time for Android and iOS using a combination of measurements with normal and power saving modes. Uplink and Downlink data rates specifications of the LTE and Wi-Fi networks with respective service providers are shown in Table I.

TABLE I. NETWORK SPECIFICATIONS

LTE	Wi-Fi
12-30 mbit/sec down	200 mbit/sec down
1-5 mbit/sec up	10 mbit/sec up
AT&T	Spectrum

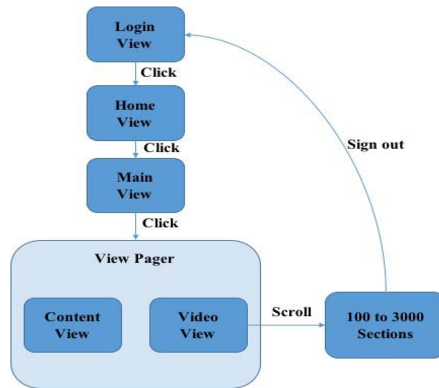

Fig. 3. Test Scenario: Educational sections and videos retrieval.

IV. EVALUATION RESULTS

This section elaborates on the measurement results from the experiments that we conducted on real android and iOS devices to evaluate our proposed performance methodology. Table II [22] shows the categories of metrics, libraries and testing frameworks used for monitoring the app interactions, saving the data to *SQLite* DB, modeling delays, implementing the video listeners for the test scenario and visualizing the effect of Firebase load on the $T_{response}$ for Android and iOS app versions. Both UIAutomator and Espresso are part of the official Android instrumentation framework. UIAutomator adds to Espresso the ability to test the GUI of the app along with the device status and performance. The combination also provides the flexibility to test multiple apps at the same time and to perform operations on the app GUI and on the device i.e. turning ON/ OFF Wi-Fi or LTE or changing the battery mode to *Power Saving* (PS) mode.

$T_{response}$ is calculated as:

$$T_{response} = T_{received} - T_{sent} \quad (1)$$

where $T_{received}$ is the timestamp of the response from Firebase is received and T_{sent} is the timestamp of request sent from the app.

TABLE II. LIST OF TESTING TOOLS AND APIS

Category	Tools
CPU utilization, Memory usage	Android profiler, Instruments (iOS)
Memory leaks	Leak detector (Leak Canary)
Testing frameworks	UIAutomator, Espresso (Android), XCTest (iOS)
Image loading and caching	Glide (Android), Nuke (iOS)
Delay modeling (*SQLite* DB)	Persistence libraries: Room (Android), SQLite.swift (iOS)
$T_{response}$	Bash, Java and Python scripts

A. Firebase Loading and Simulation Users

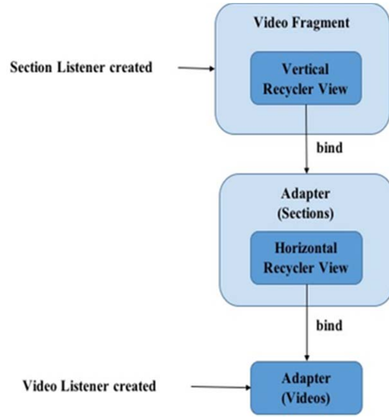

Fig. 4. Listener lifecycle

The loading of the Firebase requires simulating users and their interactions based on the test scenario discussed in the previous section. Fig. 4 shows the complete (sections + videos) listener lifecycle. It utilizes singleton pattern to keep the list of video references. These references should be removed from memory when a fragment in Android or a view controller in iOS is destroyed. The scenario is further organised as below:

1. Sections are paginated by 10 and are loaded when the user is at the bottom of the view.

2. Videos need references to section key based on their node structure in the Firebase.

Sections need to be indexed for location and to avoid duplicate listeners for the video list.

UI Automation Testing frameworks used for simulating user interactions and conducting repeatable tests were found to be sluggish and eventually crashing the app after a few number of continuous user logins. We used Android profiler and iOS Instruments tool to track the memory leaks (significantly Firebase based leaks) that are primarily the cause of Firebase listener objects not being released from memory causing a strong reference to the object the Firebase listener was created in. Hence, in order to provide consistent load to Firebase and simulating users, these strong references to the activity or view controller need to be removed by Garbage Collection (GC) and through Automatic Reference Counting (ARC) memory management mechanisms in Android and iOS respectively. If the object is no longer in use, these mechanisms recover the memory and reuse it for future object allocation. To avoid memory and performance overheads, the memory management needs to be handled during compile time rather than runtime.

While the *Usage* tab on the Firebase console offers information about simultaneous connections or users retrieving the videos by querying the complete DB node or gives a more accurate overview of DB's overall performance, we cannot really drill down enough to troubleshoot potential performance issues or fine details. With our proposed methodology, we have a flexibility to observe the performance metrics and the impact of different parameters and device settings in the app on them over time. The real-time DB (Firebase) integration with the performance module offers the deepest level of granularity as discussed in the subsections below. Performance measurements are done over a duration of 2 minutes for each of the tests as discussed next. Due to similarities in trace plots for response time with varying load in both Android and iOS, we present plots for Android LTE PS, iOS LTE Regular, Android Wi-Fi Regular, and iOS Wi-Fi PS modes.

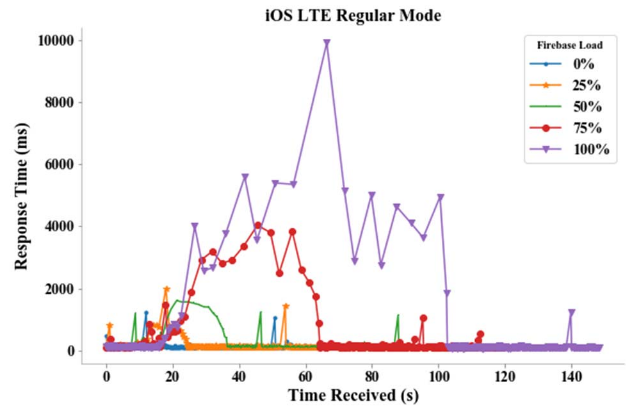

Fig. 5. Response time plot for iOS LTE Regular mode

B. LTE connection with Regular mode

In this subsection, we ran simulations on app for the test scenario with *LTE* connection and *Regular* device battery mode. Fig. 5 shows the response time plot on iOS for no load, 25%, 50%, 75% and full load on Firebase retrieving the videos. We observed that a significant load to Firebase has an immediate effect to the $T_{response}$ seen by the user. We can clearly see that response time increases with the load. For the major part of simulation duration for each load, Android and iOS perform similarly. The spikes are due to the load and the total duration of spikes for each load occur almost at the same time during the simulation in both the OS. For 100% load, the maximum peak response time and the spike duration occurs for 20 s.

C. LTE connection with Power Saving mode

In this subsection, we performed experiments with app on Android and iOS with *LTE* connection and device battery in the *Power Saving* mode. Overall, we can see the same behaviour as in section IV. B with increase in response times from no load to full load on the Firebase, with some anomalies in the 50% and 75% loads for a small period of time as shown in Fig. 6. This can be caused by inconsistent network behaviour which also explains a rapid increase in the response time from any load.

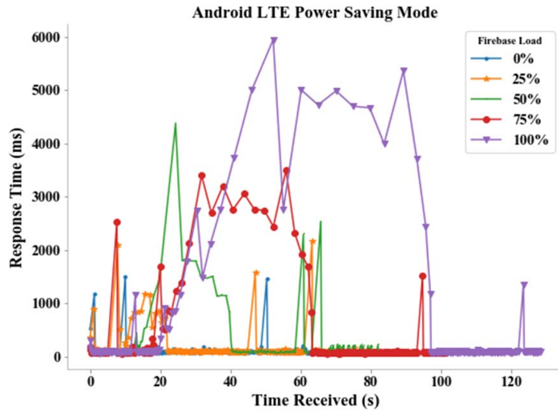

Fig. 6. Response time plot for Android LTE Power Saving mode

D. Wi-Fi connection with Regular mode

This subsection summarizes the experiments performed on app with *Wi-Fi* and *Regular* device battery mode both on Android and iOS. Here, again we observe that the response time increases with the Firebase load. However, due to noise and network inconsistencies 50% load shows a greater $T_{response}$ than 75% for initial small duration as in Fig. 7.

E. Wi-Fi connection with Power Saving mode

We further assessed the app's performance with *Wi-Fi* connection and with *Power Saving* mode. The response time as expected increases with the load in both Android and iOS (Fig. 8) versions.

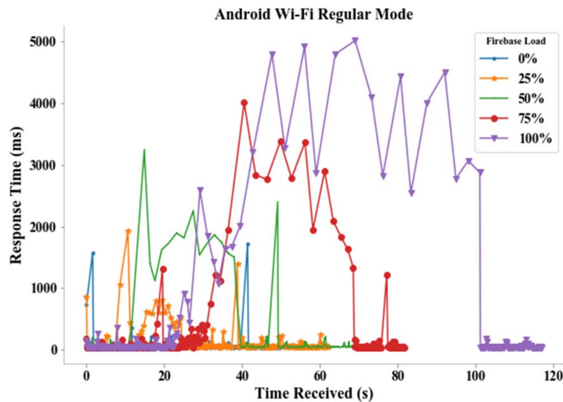

Fig. 7. Response time plot for Android Wi-Fi Regular mode

F. Android Vs iOS 95th percentile $T_{response}$

Finally, we also show the 95th percentile response times for the above covered simulation conditions on Android and iOS with Firebase loading to observe the $T_{response}$ that the majority of users will encounter excluding the outliers for a clearer picture about the characteristics of the app. In each of the graphs from Fig. 5 - Fig. 8, the response time is directly affected by the load regardless of network connectivity or device battery mode. Also, it is noted that the time to finish each load increases due to the increase of response times. The battery mode should not impact the response time due to the application running in the foreground, and the tasks to send and receive the response times are not CPU intensive. Both iOS and Android exhibit similar response times regardless of the OS. This is summarized by the 95th percentile of $T_{response}$ in Fig. 9 and 10. It is also evident that there is no single winner here. While $T_{response}$ is a key parameter to performance as expected and also as observed from the effect of Firebase loading during simulations on the UI for the test scenario, the current work also reinforces the importance of complexity of the DB queries and data modeling on the cloud DB as well as the importance of app design choices.

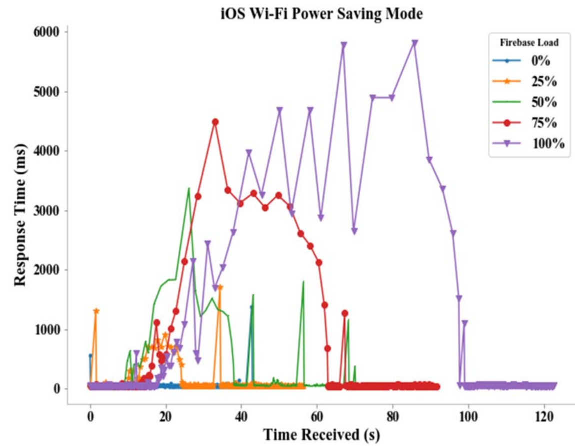

Fig. 8. Response time plot iOS Wi-Fi Power Saving mode

V. Conclusion and Future Work

This paper presents a performance assessment of an android and iOS mHealth app leveraging Google Firebase as a cloud-based testing service. In the current work, we performed a series of experiments with Android and iOS versions of the prototype HT Patient Helper app under various load conditions in the cloud-based backend DB (Firebase), with different device battery usage modes as well as mobile network connection modes. The actual trace plots of response times with respect to simulation time show that the performance is consistent in all the test campaigns for both Android and iOS and that the $T_{response}$ increases with increasing Firebase load. We found that response time performance does not appear to be correlated directly with the PS and *Regular* battery modes of operation.

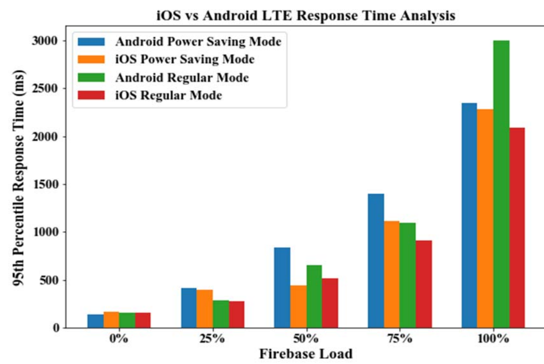

Fig. 9. iOS vs Android LTE 95th percentile Response time plot

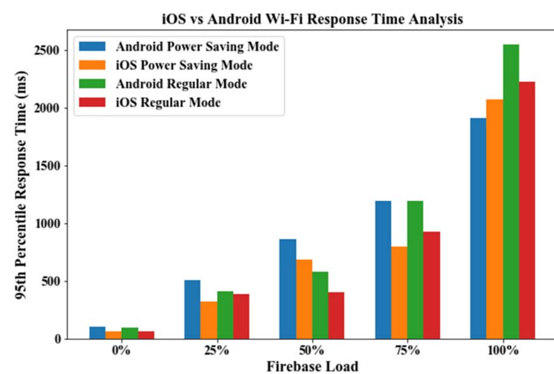

Fig. 10. iOS vs Android Wi-Fi 95th percentile Response time plot

Our study however, reveals characteristics of Wi-Fi and LTE mobile networks in the mobile-cloud communication. Wi-Fi response time measurements are at least a factor of two lower than the corresponding LTE measurements. In general, Wi-Fi is more efficient when retrieving large chunks of data if the signal strength is good. We also find that the mean $T_{response}$ measurements for Android are slightly higher than for iOS due to buffering or differences in the design of Firebase API for both. Furthermore, Firebase initially receives a large pool of requests. The spikiness in the trace plots from Fig. 5-8 for 100% load (increasing load) is due to increase in request rates while the DB is serving video retrieval requests, and there is no scaling mechanism that the current version of the app in the paper/ Firebase by default has. In the future work we plan to extend our study to investigate a broader variety of performance indicators as well in other cloud platforms at diverse geographical locations to support the performance optimization of mobile apps. We also intend to dig further the data for quantitative assessment to better understand the performance variations in the cloud data upload/ update from the app's test scenarios under the aforementioned conditions.

References

[1] M. Linares-Vasquez, K. Moran, and D. Poshyvanyk, "Continuous, Evolutionary, and Large-Scale: A New Perspective for Automated Mobile App Testing," *IEEE International Conference On Software Maintenance and Evolution,* ICSME' 17, 2017.

[2] N. Saranummi, "In the Spotlight: Health Information Systems," *IEEE Reviews in Biomedical Engineering,* vol. 1, pp. 15-17, 2008. doi: 10.1109/RBME.2008.2008217.

[3] P. Chalela, E. Munoz, D. Inupakutika, S. Kaghyan, D. Akopian, V. Kaklamani, K. Lathrop, and A. Ramirez, "Improving Adherence to Endocrine Hormonal Therapy among Breast Cancer Patients: Study Protocol for a Randomized Controlled Trial," *Contemporary Clinical Trials Communications,* vol. 12, pp. 109-115, 2018.

[4] Google, "Firebase Cloud Messaging," (2018). [Online]. Available: https://firebase.google.com/docs/cloud-messaging [Accessed Aug. 25, 2018].

[5] H. Hacigumus, B. Iyer, and S. Mehrotra, "Providing Database As a Service," in *Proceedings 18th IEEE Intl. Conf. Data Eng.,* Feb. 2002, pp. 29-38.

[6] L. Zhao, S. Sakr, and A. Liu, "A Framework for Consumer-Centric SLA Management of Cloud-Hosted Databases," *IEEE Trans. On Services Computing*, vol. 8, no. 4, pp. 534-549, Jul-Aug, 2015.

[7] C. Gyoridi, R. Gyoridi, and R. Sotoc, "A Comparative Study of Relational and Non-Relational Database Models in a Web-Based Application," *Int. J. Adv. Comput. Sci. Appl.*, vol. 6, no. 11, pp. 78-83, 2015. doi: 10.14569/IJACSA.2015.061111.

[8] M. Abourezq, and A. Idrissi, "Database-As-a-Service for Big Data: An Overview," *Int. J. Adv. Comput. Sci. Appl.,* vol. 7, no. 1, pp. 157-177, 2016. doi: 10.14569/IJACSA.2016.070124.

[9] W. Al Shehri, "Cloud Database Database As a Service," *Int. J. Database Manage. Syst.*, vol. 5, no. 2, p. 1, 2013. doi: 10.5121/ijdms.2013.5201.

[10] S.D. Bijwe and P.L. Ramteke, "Database in Cloud Computing-DataBaseas-a-Service (DBaas) With its Challenges," *Int. J. Comput. Sci. Mobile Comput.,* vol. 4, no. 2, pp. 73-79, 2015.

[11] R. Gyorodi, M.I. Pavel, C. Gyoridi, and D. Zmaranda, "Performance of OnPrem Versus Azure SQL Server," *IEEE Access*, vol. 7, pp. 15894-15902, Jan. 2019.

[12] R. Schnall, M. Rojas, S. Bakken, W. Brown, A. Carballo-Dieguez, M. Carry, D. Gelaude, J.P. Mosley, and J. Travers, "A User-Centered Model for Designing Consumer Mobile Health (mHealth) Applications (apps),"*Journal of Biomedical Informatics,* vol. 60, pp. 243-251, Apr. 2016. doi: 10.1016/j.jbi.2016.02.002.

[13] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and Detecting Resource Leaks in Android Applications," in *Proceedings 28th IEEE/ACM International Conference on Automated Software Engineering (ASE),* 2013, pp. 389-398.

[14] Y. Liu, C. Xu, and S.C. Cheung, "Characterizing and Detecting Performance Bugs for Smartphones Applications," in *Proceedings of the 36th International Conference on Software Engineering (ACM),* 2014, pp. 1013-1024.

[15] A. Nistor and L. Ravindranath, "Suncat: Helping Developers Understand and Predict Performance Problems in Smartphone Applications," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ACM),* 2014, pp. 282-292.

[16] S. Adibi, "Link Technologies and Blackberry Mobile Health (mHealth) Applications: A Review," *IEEE Trans. Information Technology in Biomedicine,* vol. 16, no. 4, pp. 586-597, 2012. doi: 10.1109/TITB.2012.2191295.

[17] H. Khalid, E. Shihab, M. Nagappan, and A. Hassan, "What Do Mobile App Users Complain About? A Study on Free iOS Apps," *IEEE Software,* vol. 32, no. 3, pp. 70-77, 2014. doi: 10.1109/MS.2014.50.

[18] Google, "UIAutomator Testing Framework," (2019). [Online]. Available: https://developer.android.com/training/testing/ui-automator [Accessed Mar. 5, 2019].

[19] Android. "Espresso Testing Framework," (2019). [Online]. Available: https://developer.android.com/training/testing/espresso/ [Accessed Feb. 20, 2019].

[20] Apple. "XCTest," (2019). [Online]. Available: https://developer.apple.com/documentation/xctest [Accessed Mar. 20, 2019].

[21] Youtube. "YouTubeLinkImporter," (2019). [Online]. Available: https://github.com/ytdl-org/youtube-dl [Accessed Apr. 10, 2019].

[22] M. Jun, L. Sheng, Y. Shengtao, T. Xianping, and L. Jian, "LeakDAF: An Automated Tool for Detecting Leaked Activities and Fragments of Android Applications," in *Proceedings of 41st Annual Computer Software and Applications Conference (COMPSAC),* vol. 1, pp. 23-32, 2017.