Mobile Applications and Unreliable Networks:
A Case Study in Standardizing Mobile Application Development

Alana Ramjit
Department of Computer Science at Columbia University

Advised by Professor Yuanyuan Zhou and Xinxin Jin
Department of Computer Science and Engineering at University of California – San Diego

Abstract

Growing pressure in a crowded mobile application market suggests that developers ought to streamline the development process in order to eliminate frustration-inducing bugs that will cause users to abandon an application. This paper presents a case-study on bugs that emerge under network stress, categorizing the most common triggers of those bugs and defining application misbehavior. It also discusses a need to organize standardized practices that can automate prevention of bugs that can be generalized to all sources of application errors.

1. Introduction

Numerous studies have pointed to the consistent trend of growth in mobile applications available on the market. A 2013 estimate from Canalys estimates that the Apple App Store and Google Play each have around 800,000 apps available each, with numbers sure to have grown in the past year.[1] This figure does not account for Blackberry and Windows apps, putting a reasonable estimate at around 2 million apps.

However, the actual usage data of these apps give significant insight as to end-users engagement and response to the growth in availability of mobile apps. Latest Nielsen reports indicate that there is an upper limit on the *number of apps per month* that users will interact with though they may spend significantly more time on those apps once downloaded.[2] Additionally, users have a very low tolerance for application failure; only 16% of users are willing to try an app more than two times if it crashes.[3]

Consequently, there is significant need to streamline application development for a better end-user experience on the first try. Most of the work on improving mobile application behavior has focused on diagnosing and reporting bugs and crashes. This paper furthers that discussion by addressing what to do when those bugs are found and how to eliminate them from occurring at early developmental stages rather than patches after release.

Grasping end-user behaviors and attitudes is important for developers seeking to maintain or introduce apps in a crowded market. Most importantly, trends in end-user behavior can and should be used to intelligently shape, standardize, and streamline the development process. The mobile app development life-cycle has emerged relatively ad-hoc, with few industry standards or best practices for testing or app behavior in place.

This paper attempts to address the community-wide issues in application development by presenting a brief case study of network connectivity related bugs and crashes in real-world mobile apps. It does the following:

(1) Provides a field test of eight applications, four from large-resource development companies and four from small, independent developers to investigate the question of *what constitutes application misbehavior?* Is it simply crashing and freezing or do testing

frameworks need to extend their analysis to find other serious common patterns of misbehavior that may cause users to leave? It also provides a survey of current practices in dealing with network stress to determine what "best practices" have been developed as well as indicating patterns of misbehavior among applications.

(2) Examining bug-fix and crash report data from open source Android applications that will showcase both the lack of and need for industry wide standards in responding to the non-trivial issue of unreliable networks. It analyzes them for network conditions which are most likely to trigger application misbehavior from an environment perspective and what the cause might be from a development perspective.

(3) This case study is meant to be a jumping-off point for further investigation into raising questions about how to standardize application behavior and streamline development in a mobile context.

2. Related Work and Background

Network connectivity stress is chosen for a number of reasons. First, it is an example of a test case that occurs frequently "in the wild" but not necessarily in test environments. Dropped networks due to a sudden move to a low-reception area, slow or busy data connections, or a switch from WiFi to 3G or vice versa can all trigger unexpected and potentially missed behaviors that would not be picked up in a test environment.

Secondly, designing and implementing a robust, standardized protocol for how to respond to dropped connectivity is not a trivial problem. It raises issues regarding ideal behavior—how long should an application wait to receive confirmation from the network? How should the main UI thread respond while waiting? If data is lost being sent, should it be automatically re-sent or alert the user?

Thirdly, this is an issue that is unique to mobile applications, and highlights the need for new testing and development practices for mobile that go beyond what exists for web and desktop applications. Not only should these questions be answered for the specific problem of network connectivity, but they should be raised about other common bugs in mobile applications.

Several other studies and papers have investigated this problem and corroborate the claim that network connectivity stress is a leading cause of mobile application failure.

A call for standardization—Dongsong Zhang and Boonlit Adapt called for a standardization of best practices of testing mobile applications similar to what has evolved for desktop applications.[4] They cite network connectivity as one of the mobile-specific issues needed to be addressed and provide a general framework for data collection and usability testing.

Demonstrating the need for standardizing network connectivity --others have attempted to either diagnose common causes of crashes and bugs in mobile applications or improve ability to diagnose these causes in the field. Most of these have focused on developing dynamic test environments rather simply analyzing static binaries in order to pinpoint, trigger, and diagnose bugs. Many of these have found that network loss and inability to handle changes in connectivity is one of the most common causes of bugs.

The first example of this is VanarSena.[5] VanarSena uses "fault injection monkeys (FIMs)" to trigger crashes internally and simulates bad external conditions such as improper user/sensor input or poor network connection. The most relevant data from this study shows that **assuming a reliable network and server or failure to handle poor network connectivity aggregated to the largest root cause of application failure.** Out of the nearly 3000 bugs uncovered, network-related

failure caused over 60% them, ranging from poor connectivity, bad or malformed data, or HTTP error codes that were unhandled.

Similarly, Caiipa is a testing framework that goes a step beyond UI automation to prioritize which real-world contexts are most relevant to a particular app, then simulates exposing the application to high stress in those contexts.[6] The results have shown a significant improvement in discovering performance bugs and crashes. This is a step forward because it anticipates the unique challenges of a mobile application: that it is mobile, and therefore liable to being exposed to a wide variety of competing environmental factors that cannot be tested by pure UI automation. Caiipa uncovered an illustrative example in which the Twitter app frequently crashes while switching from WiFi to 3 or 2G network.

Automating standards for network management—Procrastinator comes closest to realizing the potential of standardizing mobile development.[7] It presents a tool which takes existing application binary and injects custom code to solve a known problem. Specifically, it prevents mobile apps from pre-fetching data unnecessarily which can cause wasted data usage, a serious turn-off for users who pay per byte of data usage. It reformulates a standard programming practice of pre-fetching as much data as possible, adapts it to mobile-specific platforms, and then automates the process of developing for this platform. When calling for industry wide evaluation and standardization, Procrastinator serves as a base model for what can be achieved in the field of network connectivity stress as well as in other mobile contexts.

3. Methods

In order to investigate the prevalence of network connectivity induced bugs and crashes, this paper examines applications designed for the Android platform, largely because of it's open source nature that allows developers of all types to contribute applications.

3.1 Investigating the Android Platform

First, we examine the Android development library for the standard functionality it provides for guiding developers in network access. Then we break down the most common conditions under which an application might need to access the network.

3.2 Application Behavior Case Study

Secondly, we perform a case study examination of eight applications available in the GooglePlay store, four of which are highly popular applications developed by companies with presumably large amounts of resources, and four of which come from smaller developers and have fewer downloads (< 10,000). The question asked by this case study is what sort of application behavior do we expect under network stress and what actually happens?

The eight applications that were chosen for this case study, as previously stated, can be divided into two categories based on number of downloads and the resources available to the development team.

Those with large resources and high downloads chosen were: (1) QuizUp, a popular quiz game in which users play against each other to gain the highest score on a quiz in a particular theme, (2) Gmail, Google's immensely popular e-mail client, (3) Facebook Messenger, the stand-alone messaging client for social media giant Facebook, and (4) Pandora, one of the most popular internet radio streaming applications in the PlayStore.

Those with smaller resources and fewer downloads were: (1) VirginRadio, a lesser known Internet radio streaming application, (2) HoverChat, a standard SMS app designed to replace the native messaging app on Androids, (3) MailWise, an alternative e-mail handling client, and (4) Whova, a native mobile app to coordinate events and groups.

Application	Functionality	Full Network/ WiFi	No network	Weak Network (3G, 1 bar)
QuizUp	Launch	Successful	Error Message	Endless load/Error Message
	Play	Successful	Freeze/Error Message	Error Message/Lost Score
Gmail	Send Email	Successful	Error Message	Fail/Automatic Retry
	Refresh/Sync	Successful	Error Message	Interminable Load
Pandora	Launch	Successful	Error Message/Close	Erratic/Crash/Freeze
	Mid-stream	Successful	Skip songs/Endless load	Skip songs/Crash
FB Messenger	Send Message	Successful	Error message/Retry menu	Error message/Retry menu
	Launch	Successful	Successful/ Error message	Successful/ Error message

Figure 1. Table of application behavior under network stress for "large-resource" group

All of the apps were tested under three basic network conditions: strong WiFi signal as a baseline for comparison, no network connection, and a weak network connection defined as 1 or fewer bars on a 3G network. All applications were tested on the same hardware/software, a Samsung Galaxy S3. Only these three basic network conditions were tested for the sake of testing simplicity; other potential real-world situations such as a switch from a fast connection, e.g. WiFi, to a slower connection, e.g. 3G or 2G were too difficult to reliably simulate.

3.3 Open Source Issue Tracker Analysis

Previous studies discussed above have already established that network-related errors are one of the most common sources of bugs in mobile applications. In order to determine which network conditions trigger these bugs, 44 bug reports from open source applications or issue trackers that cite network-related errors are examined. From this sample, we quantify what types of network stress tend to cause bugs in practice.

4. Results

Basics of the Android Platform

We identify seven major categories of network access. These are (1) send a message or post such as to a thread or via SMS, (2) load a new page, (3) sync the cache, (4) database access such as log-in/sign-up or view high scores, (5) p2p connection such as for multi-player games or location based chats, (6) download a file or stream media, and (7) coarse GPS assistance which usually pings WiFi or network towers instead of activating location sensors. The latter of these is the least important because GPS does not necessarily require network access to function.

Within the Android development library itself, we find three major functions and classes for regulating network access. The function getSystemService() returns an object that is castable to the type ConnectivityManager, which itself is capable of retrieving network access information. The functions getActiveNetworkInfo() and isConnected() can be used to check if there is a valid network; if the former is not null and the latter is true, it generally assumed safe to retrieve data. Finally, the AsyncTask class can be extended to prevent potentially very slow network operations from occurring in the main UI thread and causing a sluggish, unresponsive app.

While several other functions and classes exist to regulate and manage network connectivity, particularly for p2p connections, these are the core functionalities provided for accessing both WiFi and network data.

Case Study

The results of field testing on "big" applications are displayed in Figure 1, and the results of testing on "small" applications in

Application	Functionality	Full Network/ WiFi	No network	Weak Network (3G, 1 bar)
Whova	Launch	Successful	Error Message/Fail	Mixed error/loading messages
	Post/Send	Successful	Error Message	Fail with Success Confirmation
Mailwise	Send Email	Successful	Error Message/Auto resend	Confirmation then error
	Refresh/Sync	Successful	Error Message	Re-pull previously read as new
Virgin Radio	Launch	Successful	Fail/No Error Message	Unreliable fail without warning
	Mid-stream	Successful	Cut off/No Error Message	Cut off/No Error Message
Hoverchat	Send Message	Successful	Error Message/Retry Menu	Error Message/Retry Menu

Figure 2. Table of results for smaller resource application behavior under network stress

Figure 2. Each app was tested for its core functionality, with benign or successful behavior defined as the application performing exactly what was intended OR the application failing, alerting the user, and doing nothing else. Furthermore, the eight applications were selected such that it had full coverage of the six core categories requiring network access.

Behavior highlighted in red indicates problematic behavior that is completely unintentional, such as repeated and intermittent freezing and crashing, automated responses that incur high data usage or battery drainage, or data loss. Orange-highlights indicates behavior that may be intentional but still problematic, such as automatically re-sending a message once the network is found. This may be problematic because, for instance, some SMS messages are time sensitive ("Meet me in 10 minutes!") and perhaps should not be sent without the user being aware.

The expectation was that larger and more popular apps would have developed a more robust stance towards handling unpredictable network connectivity. In general, this proved true. However, even within this small sample size, developers seem far better equipped to respond to the condition of no network rather weak network signals.

This emphasizes an important point regarding application development and testing. While developers know to check for an existing connection, real world environments fluctuate far more than simply on and off. These factors need to be anticipated; almost every application displayed some type of undesirable behavior under a weak signal, a common phenomenon in the daily life of users

Failing Gracefully

At a minimum, applications should notify the user that their intended interaction with the application failed and then stop. The most optimal behavior demonstrated by HoverChat and Facebook Messenger involved saving the state and data and presenting the user with a menu of options on how to proceed. The worst behaviors were those such as Pandora, which was erratic and unpredictable, QuizUp which dumped the data from a game on an unstable network, Whova and Mailwise which presented a positive verification of sending or posting that had not actually gone through, and Virgin Radio which simply failed to notify the user of anything and cut off.

Clearly, optimal application behavior is dependent on the function of the app. However, some guidelines can be pulled from this case study. Data to be sent over a network should be saved before it is lost, applications should track whether a network transfer was successful and notify users of unsuccessful transfers at a minimum. At this point, an app may gracefully exit. Optimally, the application would present the user with a menu of appropriate options depending on what the intended behavior was.

Network Triggers

The findings from the case study are verified by the larger sample of bug and patch reports examined, outlined in Figure 3 below.

Of the 44 issue reports, 29 of them were triggered when users switched between WiFi and 3G or on a slow network. Only 6 of them occurred when there was no network, indicating that developers likely anticipate the condition of no network and handle this instance appropriately, and 9 of them were present under all conditions and were likely server-side issues.

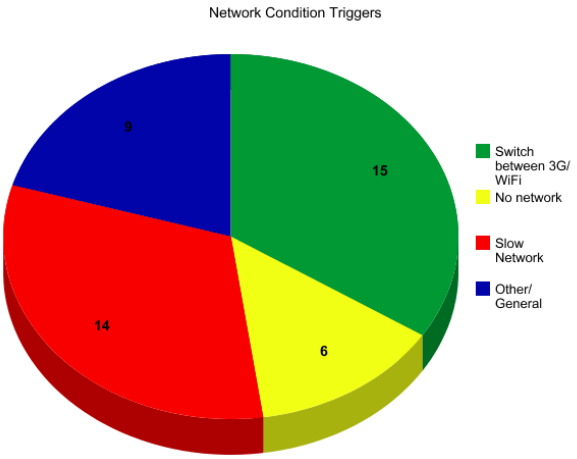

Figure 3. Most frequent network triggers

5. Conclusion and Future Work

The findings of this exploration cumulatively lead to three important insights.

First, application misbehavior is not limited to freezing and crashes and encompasses a wide range of unintended behavior. Limiting pre-launch testing to simply uncovering crash/freeze bugs potentially misses many frustrating bugs that can put off potential users. Optimal application behavior varies depending on the function of the application; nonetheless, in the case of network connectivity, it takes only a brief survey of existing practices in order to find a generalized understanding of what the minimum standard for application behavior ought to look like.

Secondly, application testing needs to fully appreciate mobile context. The most frequent triggers of bugs and worst behavior uncovered in this study came from weak signals or network switches that are incredibly common for the average user using a mobile device but are not necessarily producible in laboratory testing.

Thirdly, the most reasonable solution for eliminating network connectivity bugs at the root appears to be expanding standard library functions to include testing for and alerting of network switches or weak signals. The current Android platform framework and training tools provide functionality for checking whether or not a network connection is available and encourage boilerplate null checks for an available connection. Consequently, it is simple for developers to take advantage of existing library functions and respond to the condition of absent networks. A similar functionality and boilerplate for weak and reset connections would likely encourage better standard practice for those contexts as well.

Developing a robust and responsive framework for network access is the next step from this case study; likewise, repeating this analysis for other common sources of bugs and user frustration such as battery usage, bad user input, or malformed sensor data in order to find the access point for developing standardized best practices would also be fruitful.

References

[1] "Top IOS and Android Apps Largely Absent on Windows Phone and BlackBerry 10."*Canalys*. N.p., 23 May 2013. Web. 26 Sept. 2014.

[2] "Smartphones: So Many Apps, So Much Time."Nielsen, 1 July 2014. Web. 26 Sept. 2014.

[3] Perez, Sarah. "Users Have Low Tolerance For Buggy Apps – Only 16% Will Try A Failing App More Than Twice | TechCrunch." *TechCrunch*. N.p., 12 Mar. 2013. Web. 26 Sept. 2014.

[4] Zhang, Dongsong, and Boonlit Adipat. "Challenges, methodologies, and issues in the

usability testing of mobile applications." *International Journal of Human-Computer Interaction* 18.3 (2005): 293-308.

[5] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. In Proc. of ACM MobiSys, 2014.

[6]Liang, Chieh-Jan Mike, et al. "Caiipa: automated large-scale mobile app testing through contextual fuzzing." *Proceedings of the 20th annual international conference on Mobile computing and networking*. ACM, 2014.

[7] Ravindranath, Lenin, et al. "Procrastinator: Pacing Mobile Apps' Usage of the Network." *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 2014.