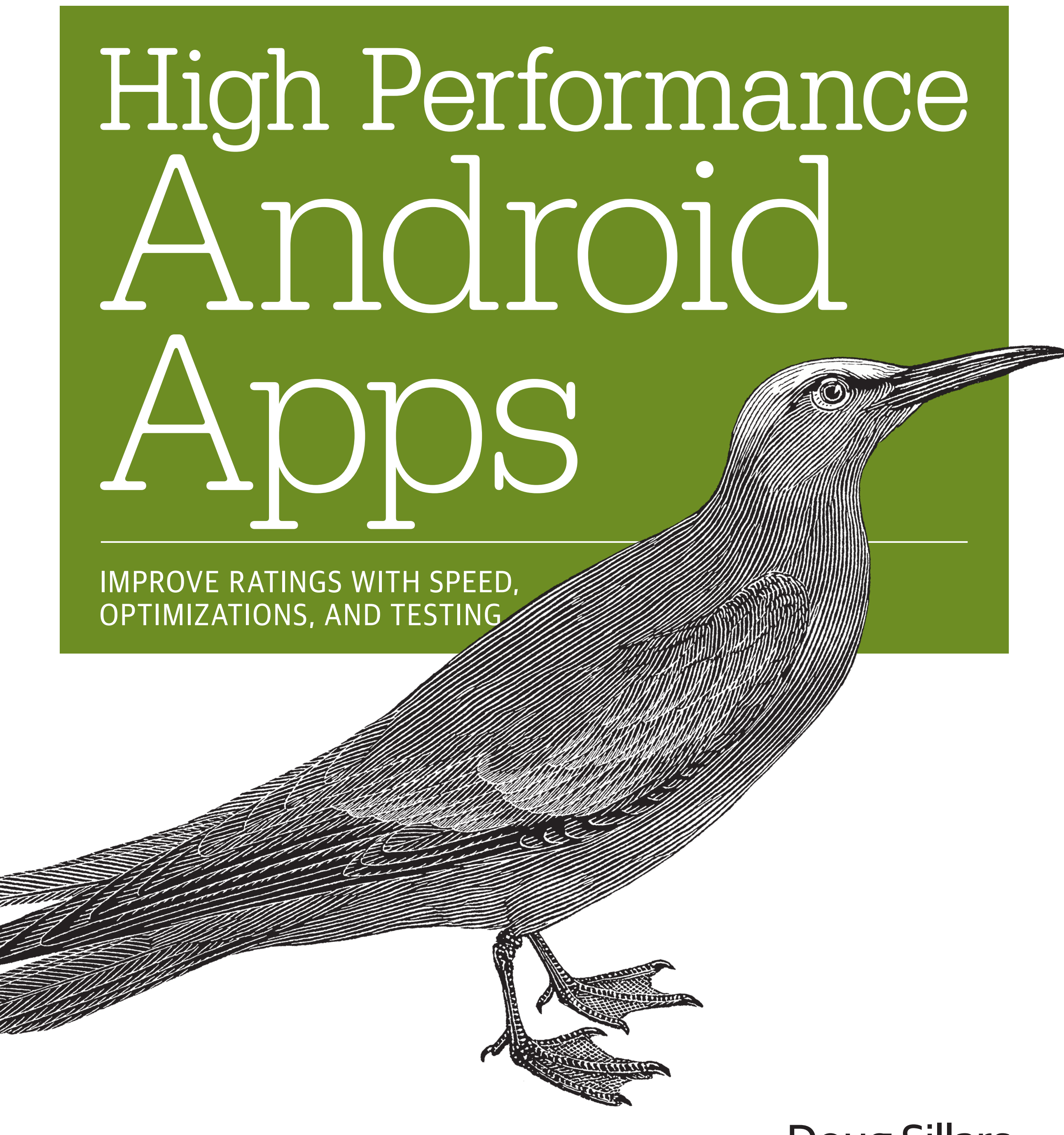
O'REILLY®

High Performance Android Apps

IMPROVE RATINGS WITH SPEED, OPTIMIZATIONS, AND TESTING



Doug Sillars

www.allitebooks.com

High Performance Android Apps

Unique and clever ideas are important when building a hot-selling Android app, but the real drivers for success are speed, efficiency, and power management. With this practical guide, you'll learn the major performance issues confronting Android app developers, and the tools you need to diagnose problems early.

Customers are finally realizing that apps have a major role in the performance of their Android devices. Author Doug Sillars not only shows you how to use Android-specific testing tools from companies including Google, Qualcomm, and AT&T, but also helps you explore potential remedies. You'll discover ways to build apps that run well on all 19,000 Android device types in use.

- Understand how performance issues affect app sales and retention
- Build an Android device lab to maximize UI, functional, and performance testing
- Improve the way your app interacts with device hardware
- Optimize your UI for fast rendering, scrolling, and animations
- Track down memory leaks and CPU issues that affect performance
- Upgrade communications with the server, and learn how your app performs on slower networks
- Apply Real User Monitoring (RUM) to ensure that every device is delivering the optimal user experience

Doug Sillars, the performance outreach lead at the AT&T Developer Program, has helped thousands of mobile developers apply performance best practices. The tools and best practices developed at AT&T help developers make mobile apps run faster, while using less data and battery power.

"This book will empower any Android developer to produce highly efficient, well-functioning applications."

—**Brad Zeschuk**
VP Engineering, M2Catalyst

MOBILE

US $44.99 CAN $51.99
ISBN: 978-1-491-91251-5

Twitter: @oreillymedia
facebook.com/oreilly

www.allitebooks.com

High Performance Android Apps

Improve Ratings with Speed, Optimizations, and Testing

Doug Sillars

Beijing • Boston • Farnham • Sebastopol • Tokyo

www.allitebooks.com

High Performance Android Apps
by Doug Sillars

Copyright © 2015 AT&T Services, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian Anderson and Courtney Allen
Production Editor: Shiny Kalapurakkel
Copyeditor: Jasmine Kwityn
Proofreader: Elise Morrison
Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

September 2015: First Edition

Revision History for the First Edition
2015-09-04: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491912515* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *High Performance Android Apps,* the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-91251-5

[LSI]

www.allitebooks.com

Table of Contents

Foreword ix

Preface xi

1. Introduction to Android Performance 1

Performance Matters to Your Users 2
Ecommerce and Performance 2
Beyond Ecommerce Sales 4
Performance Infrastructure Savings 4
The Ultimate Performance Fail: Outages 4
Performance as a Rolling Outage 6
Consumer Reaction to Performance Bugs 7
Smartphone Battery Life: The Canary in the Coal Mine 8
Testing Your App for Performance Issues 9
Synthetic Testing 10
Real User Monitoring (RUM) 10
Conclusion 10

2. Building an Android Device Lab 11

What Devices Are Your Customers Using? 12
Device Spec Breakdown 12
Screen 12
SDK Version 13
CPU/Memory and Storage 13
What Networks Are Your Customers Using? 13
Your Devices Are Not Your Customers' Devices 14
Testing 15
Building Your Device Lab 16

www.allitebooks.com

You Want $X,000 for Devices? 16
So What Devices Should I Pick? 18
Beyond Phones 20
Android Open Source Project Devices 20
Other Options 22
Additional Considerations 23
My Device Lab 24
Conclusion 25

3. Hardware Performance and Battery Life. 27
Android Hardware Features 27
Less Is More 28
What Causes Battery Drain 29
Android Power Profile 30
Screen 32
Radios 33
CPU 33
Additional Sensors 34
Get to Sleep! 35
Wakelocks and Alarms 35
Doze Framework 37
Basic Battery Drain Analysis 38
App-Specific Battery Drain 41
Coupling Battery Data with Data Usage 44
App Standby 47
Advanced Battery Monitoring 47
batterystats 47
Battery Historian 52
Battery Historian 2.0 62
JobScheduler 66
Conclusion 71

4. Screen and UI Performance. 73
UI Performance Benchmarks 73
Jank 74
UI and Rendering Performance Updates in Android 74
Building Views 75
Hierarchy Viewer 77
Asset Reduction 90
Overdrawing the Screen 90
Testing Overdraw 91
Overdraw in Hierarchy Viewer 94

www.allitebooks.com

Overdraw and KitKat (Overdraw Avoidance) 96
Analyzing For Jank (Profiling GPU Render) 97
GPU Rendering in Android Marshmallow 100
Beyond Jank (Skipped Frames) 102
Systrace 103
Systrace Screen Painting 106
Systrace and CPU Usage Blocking Render 113
Systrace Update—I/O 2015 115
Vendor-Specific Tools 117
Perceived Performance 117
Spinners: The Good and the Bad 117
Animations to Mask Load Times 118
The White Lie of Instant Updates 118
Tips to Improve Perceived Performance 119
Conclusion 119

5. Memory Performance. 121
Android Memory: How It Works 121
Shared Versus Private Memory 122
Dirty Versus Clean Memory 122
Memory Cleanup (Garbage Collection) 123
Figuring Out How Much Memory Your App Uses 126
Procstats 131
Android Memory Warnings 136
Memory Management/Leaks in Java 137
Tools for Tracking Memory Leaks 138
Heap Dump 138
Allocation Tracker 140
Adding a Memory Leak 142
Deeper Heap Analysis: MAT and LeakCanary 145
MAT Eclipse Memory Analyzer Tool 145
LeakCanary 153
Conclusion 156

6. CPU and CPU Performance. 157
Measuring CPU Usage 158
Systrace for CPU Analysis 160
Traceview (Legacy Monitor DDMS tool) 163
Traceview (Android Studio) 166
Other Profiling Tools 170

www.allitebooks.com

Conclusion 172

7. **Network Performance.** 173
Wi-Fi versus Cellular Radios 174
Wi-Fi 174
Cellular 174
RRC State Machine 176
Testing Tools 179
Wireshark 180
Fiddler 181
MITMProxy 183
AT&T Application Resource Optimizer 183
Hybrid Apps and WebPageTest.org 187
Network Optimizations for Android 187
File Optimizations 188
Text File Minification (Souders: Minify JavaScript) 190
Images 191
File Caching 193
Beyond Files 196
Grouping Connections 196
Detecting Radio Usage in Your App 199
All Good Things Must Come to An End: Closing Connections 200
Regular Repeated Pings 202
Security in Networking (HTTP versus HTTPS) 203
Worldwide Cellular Coverage 203
CDNs 204
Testing Your App on Slow Networks 205
Emulating Slow Networks Without Breaking the Bank 206
Building Network-Aware Apps 207
Accounting for Latency 210
Last-Mile Latency 211
“Other” Radios 211
GPS 211
Bluetooth 212
Conclusion 213

8. **Real User Monitoring.** 215
Enabling RUM Tools 216
RUM Analytics: Sample App 217
Crashing 218
Examining a Crashlytics Crash Report 220
Usage 225

www.allitebooks.com

Real-Time Information 230
Big Data to the Rescue? 231
RUM SDK Performance 231
Conclusion 233

A. Organizational Performance. 235

Index. 241

www.allitebooks.com

www.allitebooks.com

Foreword

For the majority of Android Developers out there, the concept of *performance* is the last thing on their minds. Most app development is a mad sprint towards getting features in, making the UI look perfect, and figuring out a viable monetization strategy. But, application performance is a lot like the plumbing in your house; When it's working great, no one notices, or thinks about it... but when something's wrong, suddenly everyone is in trouble.

You see, users notice bad performance before any of the other features in your app. Before your social widgets, awesome image filters, or how one of your supported languages is Klingon. And guess what, users unhappy with performance, give bad reviews at a higher percentage than any other problems in your app.

This is why we say that #PERFMATTERS. It's easy to lose sight of performance as you're developing your app, but frankly, it's involved with everything you do; when users feel bad performance, they complain about bad performance, they uninstall your app, and then vengefully give you a bad review! When you think of it this way, performance sounds more like a feature that you should focus on, rather than a burden you have to put up with.

But in all honesty, improving performance is a really tough thing to do. It's not enough to understand your algorithm, you need to understand how Android is responding to it, and then how the hardware is responding to Android. The truth is, that one line of code can trash the performance of your entire app, simply because it's abusing some hardware limitation. But you can't stop there, because in order to even understand what's going on under the hood, you have to learn a whole separate set of tools that are built just for performance profiling. Basically, it's an entirely new way of looking at application development, and it's not for the faint of heart.

But that's what's so great about the book that Doug has put together here. It's the 'in the trenches' guide to everything performance on Android. Not only does it cover the basic algorithm topics, but also goes into how the hardware and platform are working so you

can understand what the crazy tools are telling you. This is the type of book that helps to transform an engineer's perspective of the platform. It stops being about views and event listeners, and slowly grows to an understanding of memory boundaries and threading problems.

When it's 4am, your app is running poorly, the coffee machine is out, and your startup incubator room smells like cabbage; this is the book you'll crack open to make sure that 10:00 AM meeting with the Venture Capitalists runs smoothly.
Good luck!

—Colt McAnlis, Senior Staff Developer Advocate, Google Inc.Team Lead, Android Performance Patterns - https://goo.gl/4ZJkY1

Preface

You are building an Android application (or you already have). Despite this, you are not totally happy with your app's performance (why else did you pick up this book?). Uncovering mobile performance issues is a job that is never complete. In my research, I found that 98% of apps tested had room for potential performance improvements. This book will cover the pitfalls of mobile performance and introduce you to some of the tools to test for issues. My goal is to help you acquire the skills necessary for catching any major performance issues in your mobile app before they impact your customers.

Studies have shown that customers *expect* mobile apps to load quickly, rapidly respond to user interactions, and be smooth and pleasing to the eye. As apps get faster, user engagement and revenue increase. Mobile apps built without an eye on performance are uninstalled at the same rate as those that crash. Apps that inefficiently use resources cause unnecessary battery drain. The number one complaint carriers and device manufacturers hear from customers concerns battery life.

I have spoken to thousands of developers about Android app performance over the last few years, and few developers were aware of the tools available for solving the issues they experience.

The consensus is clear: mobile apps that are fast and run smoothly are used more often and make more money for developers. With that information, it is surprising that more developers are not using the tools that are available to diagnose and pinpoint performance issues in their apps. By focusing on how performance improvements affect the user experience, you can quickly identify the return on investment that your performance work has made on your mobile app.

Who Should Read This Book

This book covers a wide range of topics centering around Android performance. Anyone associated with mobile development will appreciate the research around app performance. Developers of non-Android mobile apps will find the arguments and issues around app performance useful, but the tools used to isolate the issues are Android specific.

Testers will find the tutorials of tools used to test Android performance useful as well.

Why I Wrote This Book

There is a large and burgeoning field of web performance in which developers share tips on how to make the Web fast. Steve Souders wrote *High Performance Web Sites* in 2007 (O'Reilly), and the topic is covered in books, blogs, and conferences.

Until recently, there has been very little focus on mobile app performance. Slow apps were blamed on the OS or the cellular network. Poor battery life was blamed on device hardware. As phones have gotten faster and the OSs have matured, customers are realizing that mobile apps have a role in the performance of their phones.

There are many great tools for measuring Android app performance, but until now, there hasn't been a guide listing them all in one place. By bringing in tools from Google, Qualcomm, AT&T, and others, I hope this book will take some of the mystery out of Android performance testing, and help your app get faster while not killing your customers' batteries.

Navigating This Book

When it comes to studying application performance, I have chosen to look at how your app's code affects different aspects of the Android device. We'll start at a high level: performance and the Android ecosystem, and then look at how your app's behavior affects the screen, CPU, network stack, etc.

Chapter 1, Introduction to Android Performance
: This chapter introduces the topic of mobile app performance. We'll run the numbers to show how crucial performance is to your app. I'll highlight many of the challenges, but also the effects of poor performance in the marketplace. These are the stats you can use to convince your management that putting effort into speeding up your apps is time well spent. The data presented here generally holds for all mobile platforms and devices.

Chapter 2, Building an Android Device Lab
: Here we'll cover testing. Android is a huge ecosystem with tens of thousands of devices, each with different UIs, screens, processors, and OS versions (to name just a few considerations). I'll walk through some of the ideas to help your testing cover as many device types as possible without breaking the bank (too much).

Chapter 3, Hardware Performance and Battery Life
: Next, we'll discuss the battery, including what causes drain and how much drain. In addition, this chapter covers how your customers may discover battery issues in your app, and developer tools to isolate battery issues. We'll also look at the new JobScheduler API (released in Lollipop), which abstracts application wake-ups to the OS.

Chapter 4, Screen and UI Performance
: Screen performance accounts for the largest power drain on users' phones, and the screen serves as the primary interface to your app—this is where slow apps show jank (skipped frames) and slow rendering. This chapter walks through the steps to optimize the UI by making the hierarchy flatter, and how to test for jank and jitter in your app using tools like Systrace.

Chapter 5 and Chapter 6, Memory Performance and CPU and CPU Performance
: These chapters look at memory and CPU issues such as garbage collection, memory leaks, and how they affect the performance of your app. You'll learn how to dig into your app to discover potential issues by using testing tools such as Procstats, Memory Analysis Tool (MAT), and Traceview.

Chapter 7, Network Performance
: Here we'll look at the network performance of your app. This is where I got started in mobile performance optimization, and we'll look into the black box of how your app is communicating with your servers and how we might enhance these communications. We'll also look at how to test the performance of your app on slower networks (as much of the developing world will be on 2G and 3G for decades to come).

Chapter 8, Real User Measurements
: Finally, we'll discuss how to use real user-monitoring and analytics data to ensure that every device is getting the optimal user experience. As shown in Chapter 2, there is no way to test every Android device out there, but it is up to you to *monitor* the performance of your app on your customers' devices.

Appendix A, Organizational Performance
: Here we'll cover organizational performance, including how to get buy-in to build performant apps. By sharing the research, success stories, and proofs of concept, you can show your company that placing performance as a goal for the whole organization will improve the bottom line.

Using Code Examples

There are several sample apps in the book. The sample code can be found at *https://github.com/dougsillars/HighPerformanceAndroidApps*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*High Performance Android Apps* by Doug Sillars (O'Reilly). Copyright 2015 AT&T Services, Inc., 978-1-491-91251-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Conventions Used in This Book

The following typographical conventions are used in this book:

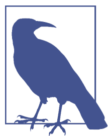
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.



This element signifies a general note.

This element indicates a warning or caution.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/0636920035053.do*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

To everyone at AT&T—my boss, Ed (and his boss, Nadine), and everyone in the AT&T Developer Program (Ed S., Jeana, and Carolyn). I would like to especially shout out to the ARO team members—Jen, Bill, Lucinda, Tiffany Pete, Savitha, John, and Rod (and of course all the devs!)—who work every day to share performance tools with the developer community. My colleagues past and present in AT&T Labs: Feng, Shubho, Oliver—thank you for coming up with the idea of ARO, and getting us involved in app performance.

A big thank you to everyone who read the early iterations of the book—your comments, tips, and suggestions were invaluable. To my technical reviewers and editors—thank you for all the great feedback and suggestions. You have helped make the book stronger!

Last, but most importantly, I would like to thank my wife and three children for their patience through the late nights (and the subsequent grumpy mornings) as this book grew from an idea to final form. I couldn't have done it without you guys, and I love you so very much. 1437313.

It's funny—I have a PhD studying chemical reaction mechanics and kinetics (how reactions work, and how to make them faster). Who knew that it would translate into a career studying mobile app mechanisms, optimizations, and kinetics?

CHAPTER 1
Introduction to Android Performance

Performance can mean a lot of different things to different people. When it comes to mobile apps, performance can describe how an app works, how efficiently it works, or if it was enjoyable to use. In the context of this book, we are looking at performance in terms of efficiency and speed.

From an Android perspective, performance is complicated by thousands of different devices, all with different levels of computing power. Sometimes just getting your app to run across your top targeted devices feels like an accomplishment on its own. In this book, I hope to help you take your app a step further, and make it run *well* on 19,000 different Android devices, giving *every* user the ultimate experience for your Android app.

In this book, we will look at app performance specifically in terms of power management, efficiency, and speed. We will cover the major issues mobile app developers face, and explore tools that will help us identify and pinpoint performance issues typically found in all mobile apps. Once the tools help us isolate the issues, we'll discuss potential remedies.

This book should be useful to anyone whose team is developing Android apps. Performance leads, single developers, and teams of developers and testers will all find benefits from the various performance tools and techniques discussed in the following chapters.

As with all suggestions to make your code optimized, your mileage may vary. Some fixes will be quick and easy wins. Other ideas may require more work, code refactoring, and potentially major architectural changes to your mobile app. This may not always be feasible, but knowing where your app's weaknesses are can help you as you iterate and improve your mobile app over time.

By learning the techniques to benchmark the performance of your app, you will be ready to profile when you feel like there is an issue. Knowing the tricks to improve the efficiency, performance, and speed of your app can help you avoid slowdowns and customer complaints.

Performance Matters to Your Users

How fast does your app have to be? Human engagement studies (going back to the 1960s) have shown that actions that take under 100 ms are perceived as instant, where actions that take a second or more allow the human mind to become distracted.[1] Delays and slowness in your app (even if just the *perception* of slowness) is probably one of the biggest killers of app engagement out there. It can also potentially damage your customers' phones (a study in 2012 found that slow apps caused 4% of users to throw their phone![2]).

Ecommerce and Performance

Imagine an ecommerce app that has collected analytics showing the average shopping session is 5 minutes long, and each screen load takes an average of 10 seconds to complete. Your screen view budget per session is 30 views to complete a sale. If you are able to lower the load time of each view by 1 second, you have added 3 more screen views to the average session. This could allow your customers to add more items to their cart, or perhaps just complete the entire transaction 30 seconds faster!

This completely made up scenario is actually backed by real-world data. A study in 2008 on desktop websites show that slower websites have fewer page views/sales and lower customer satisfaction than faster sites.[3]

In fact, my fictional ecommerce site improvements match the Figure 1-1 data exactly. By adding 3.3 page views to a session, we added 11% more page views!

1 Jakob Nielsen, "Response Times: The 3 Important Limits," excerpt from Usability Engineering (1993), *http://www.nngroup.com/articles/response-times-3-important-limits/*.

2 Mobile Joomla!, "Responsive Design vs Server-Side Solutions," December 3, 2012, *http://www.mobilejoomla.com/blog/172-responsive-design-vs-server-side-solutions-infographic.html*.

3 Roger Dooley, "Don't Let a Slow Website Kill Your Bottom Line," Forbes, December 4, 2012, *http://www.forbes.com/sites/rogerdooley/2012/12/04/fast-sites/*.

www.allitebooks.com

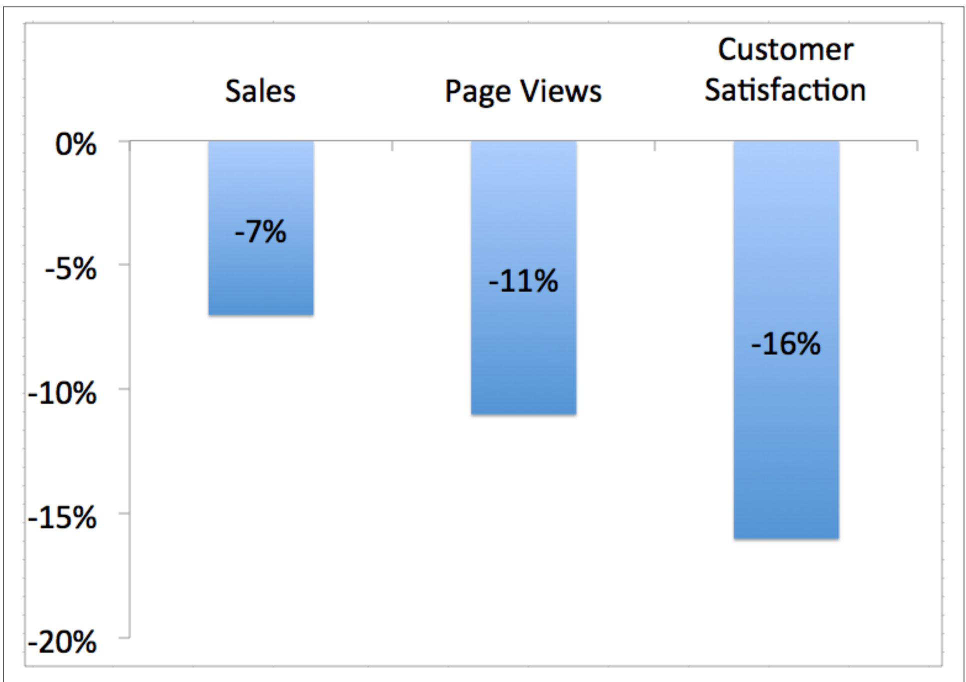


Figure 1-1. Effects of slow websites (based on a 1-second web page delay)

Performance studies on web performance provide a lot of context to mobile app performance. There are many studies showing that speeding up website performance increases engagement and sales. I would argue that all desktop performance results hold for mobile (and due to the *instant gratification* of mobile, they may even be low estimates).

Amazon[4] and Walmart[5] have independently reported similar statistics. Both major retailers found that just 100 ms of delay on their desktop web pages caused their revenue to drop by 1%. Shopzilla rearchitected its website for performance, and saw page views increase by 25% and conversions increase by 7%–12%,—all while using half the nodes previously required![6]

4 Todd Hoff, "Latency Is Everywhere And It Costs You Sales - How To Crush It," High Scalability, July 25, 2009, *http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it*.

5 Joshua Bixby, "4 Awesome Slides Showing How Page Speed Correlates to Business Metrics at Walmart.com," Radware, February 28, 2012, *http://www.webperformancetoday.com/2012/02/28/4-awesome-slides-showing-how-page-speed-correlates-to-business-metrics-at-walmart-com/*.

6 Todd Hoff, "Latency Is Everywhere."

Beyond Ecommerce Sales

In addition to decreased sales and revenue, mobile apps with poor performance receive lower rankings in Google Play. Even worse are the stories of badly behaved apps being pulled from consumer devices. In 2011, T-Mobile asked Google to remove YouMail, a third-party voicemail app, from what was then called the Android Market. The way YouMail checked for new voicemails on the server was to wake up the device and poll at 1-second intervals (yes, that's 3,600 pings/hour)! This frequent connection caused an install base of ~8,000 customers to generate more connections on the network than Facebook! Arguably, this all occurred prior to widespread usage of Google Cloud push messaging. But apps with similar behavior are still in Google Play today, and as we will see, they have detrimental performance effects on servers, networks, and most importantly—our customers' Android devices.

Sometimes your architecture is *good enough* for launch, but what happens when you get bigger? What if your app gets an ad placed during the next Super Bowl? Is your app/server architecture ready for fast exponential growth?

Performance Infrastructure Savings

Most Android apps are highly interactive and download a lot of content from remote servers. Lowering the number of requests (or reducing the size of each request) can yield huge speed improvements inside your app, but it will also yield huge reductions in traffic on your backend—allowing you to grow your infrastructure at a less rapid (expensive) pace. I have worked with companies that have reduced the number of requests by 35%–50% and the data traffic by 15%–25%. By reducing the work being done remotely, millions of dollars per year were saved.

The Ultimate Performance Fail: Outages

A study of Fortune 500 companies has shown that in 2015, website outages cost companies between $500,000–$1,000,000 per hour. In addition to loss of revenue, there are costs to bring data centers, cloud services, databases, etc. back up. Looking back over the last decade, there have been multiple studies[7] estimating the costs of an outage (and they are rising). Two of these studies attribute 35%–38% of outage costs to lost revenue. If we apply that model to all of the studies, we find that a one hour out-

7 Yevgeniy Sverdlik, "One Minute of Data Center Downtime Costs US$7,900 on Average," DatacenterDynamics, December 4, 2013, *http://www.datacenterdynamics.com/critical-environment/one-minute-of-data-center-downtime-costs-us7900-on-average/83956.fullarticle;*
Martin Perlin, "Downtime, Outages and Failures - Understanding Their True Costs," Evolven, September 18, 2012, *http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html;*
AppDynamics, "DevOps and the Cost of Downtime: Fortune 1000 Best Practice Metrics Quantified," *http://info.appdynamics.com/DC-Report-DevOps-and-the-Cost-of-Downtime.html*.

age causes a $175K loss in revenue per hour in 2015, and the costs are just getting higher (Figure 1-2).

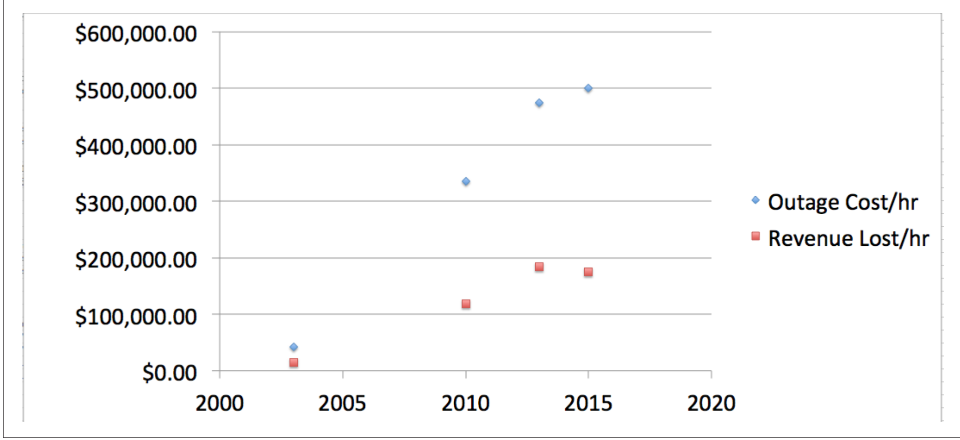


Figure 1-2. Cost of an outage per hour

An outage is certainly the worst type of performance issue. Nothing works! For this reason, companies spend millions of dollars a year to prevent complete outages of their content. In addition to loss in revenue and customer satisfaction, when there is an outage, our customers also have no idea how to react (see Figure 1-3).

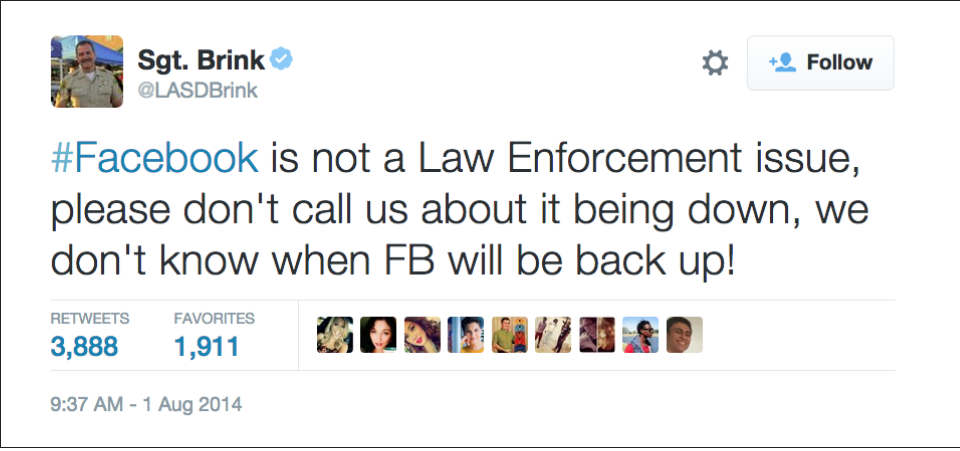


Figure 1-3. Sgt. Brink of the Los Angeles County Sheriff's Department tweeted an advisory in response to a high volume of inquiries regarding a Facebook outage

But in all seriousness, uptime performance is crucial to the survival of your company. The mobile analogy to an outage is when your app crashes. Obviously, the first performance issue you must resolve are crashes, because if the app doesn't work, it

doesn't matter how fast it is. However, when your app is running slower or even *appears* to be slower, your customers will have a similar reaction to when there is an outage.

Performance as a Rolling Outage

When there is a brownout, your electric company is providing a lower voltage, and your lights appear dim (and your fridge might stop working altogether). A slow Android app operates in the same way. Your customers can still use your app, but scrolling may be laggy, images may be slow to load, and the whole experience *feels* slow. Just as a brownout adversely affects an electric company's customers, a slow Android app is equivalent to a rolling ongoing outage. In March 2015, HP published "Failing to Meet Mobile App User Expectations", a study that shows customers react to slow apps the same way they do to apps that crash (Figure 1-4).

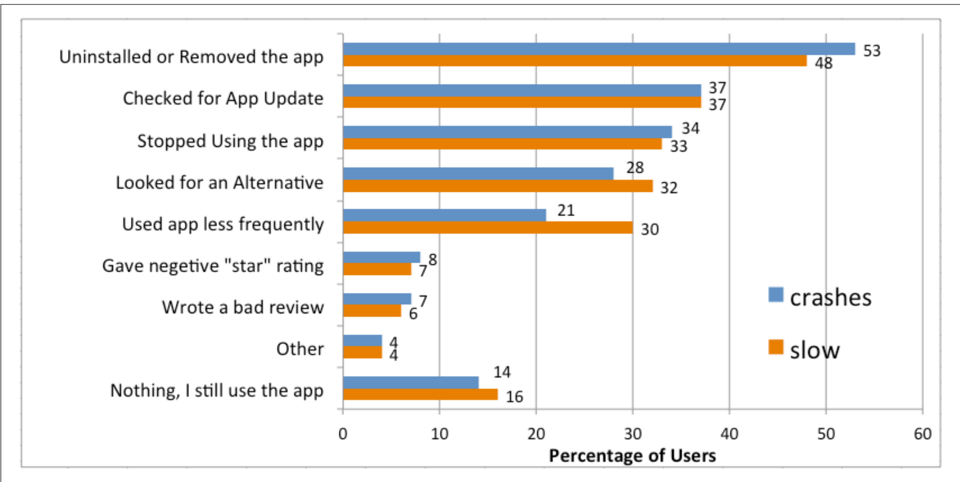


Figure 1-4. Poor performance versus crashes: same consumer result

If we cross reference the data from the "Performance Matters to Your Users" on page 2 section with the data from the "The Ultimate Performance Fail: Outages" on page 4 section, we can come up with estimates of the cost of slow performance (Figure 1-5). When there is an outage, your app loses revenue.[8] If we know that after 4.4 s of load time, conversions drop 3.5%–7%, we can estimate that a slow "rolling outage" costs your bottom line as much as $6K–$12K per hour.

8 Some customers will come back and buy later, so this is a low estimate of revenue per hour.

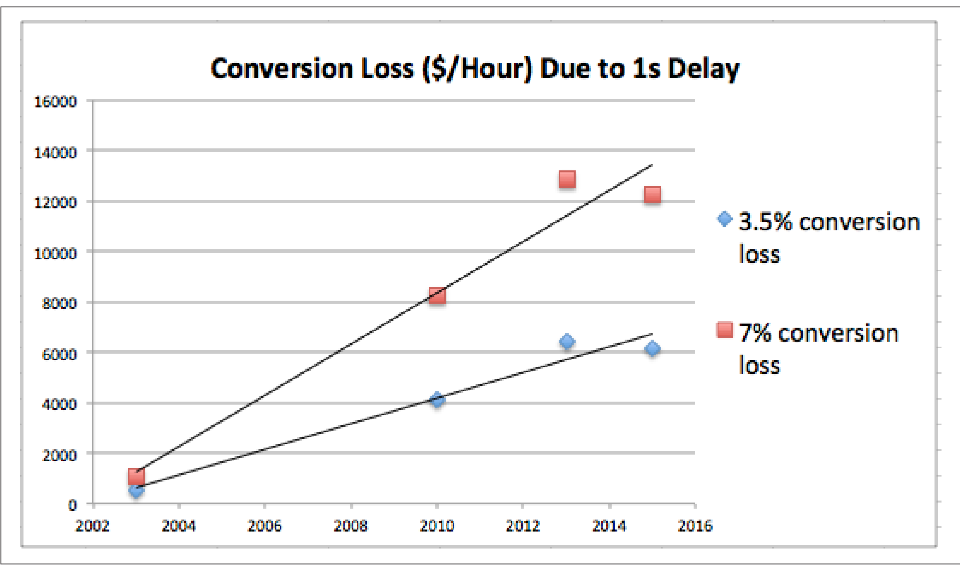


Figure 1-5. Cost of a slow app per hour (based on a 1-second web page delay)

As Figure 1-5 shows, the cost of a slow app is increasing year over year. As your app loses revenue and customers, eventually your revenue will drop to zero—which is something I hope never happens to your app.

Consumer Reaction to Performance Bugs

With such a complicated development platform, it is inevitable that some bugs will slip through your testing processes and affect customers. A recent study showed that 44% of Android app issues and bugs were discovered by users, and 20% of those were actually reported to developers by users who submitted feedback in Google Play reviews.[9] Negative reviews are not the way you want to discover issues. Not only is one customer frustrated, but all of your future potential customers will have the ability to see your dirty laundry when they look at the reviews. When customers see reviews discussing bugs and problems with your app, they may decide to not continue the download. Anything that prevents your app from being downloaded is costing you money!

Once the app has been downloaded, you are not out of the woods. In 2014, 16% of downloaded Android apps were launched only once.[10] Customers are easily distrac-

9 Perfecto Mobile, "Why Mobile Apps Fail: Failure to Launch," Fall 2014, *http://info.perfectomobile.com/rs/perfectomobile/images/why-mobile-apps-fail-report.pdf*.

10 Dave Hoch, "App Retention Improves - Apps Used Only Once Declines to 20%," Localytics, June 11, 2014, *http://info.localytics.com/blog/app-retention-improves*.

ted, and with so many choices in the app markets, if your app doesn't satisfy, they will quickly try a similar app. Although there are many reasons why users abandon apps, it can be argued that being frustrated with an app is a top reason to abandon or uninstall. According to a study by Perfecto Mobile,[11] the top user frustrations are as follows:

- User interface issues (58%)
- Performance (52%)
- Functionality (50%)
- Device compatibility (45%)

While performance is directly called out as the number two reason for customer frustration, it is clear that the other responses also have aspects of performance to them. It becomes pretty clear that the major reasons customers stop using apps are due to issues related to performance.

Adopting a minimum viable product (MVP) approach to your Android app—where the initial launch contains bugs and performance sinks—assumes that when the fixes are made, you:

- Still have an audience
- They update your app
- They launch the updated app to see the improvements

Twitter has reported that it takes 3 days for 50% of users to upgrade its Android app, and 14 days for 75% of users to update to the latest version.[12] The company found this to be extremely repeatable. So if your app is not uninstalled, you still have to hope that your updates (with all your fixes and improvements) are:

- Actually downloaded by users
- Opened up so that the fixes are seen

Smartphone Battery Life: The Canary in the Coal Mine

The studies discussed in the previous section show that consumers prefer fast apps, and apps that do things quickly. One of the top concerns of smartphone owners is battery life. While it is not (yet) common knowledge to customers, apps (and especially non-optimized ones) can be a *major* factor in battery drain. I use my end-of-

11 See the "Why Are Your Mobile Apps Failing?" infographic.

12 See "Scaling Android Development at Twitter", a presentation by Twitter's Jan Chong at Droidcon Paris 2014.

day battery percentage as an indicator of how apps are performing on my phone. If I notice a sudden dip in battery life, I begin to investigate recently downloaded apps for potential issues.

One common refrain in new Android device reviews is that “battery life is not improved from <previous model>.” My contention is that if these reviewers set up their new devices with the same apps that were on their previous devices, battery life would be similar. One of his or her cherished apps that was moved to the new phone is killing the battery. In Chapter 3, we’ll show how battery drain of the mobile device can be used as a proxy for application performance, and how improving the performance of your app will extend battery life for your customers.

The top drainers of mobile battery are the screen, the cellular and Wi-Fi radios, and other transmitters (think Bluetooth or GPS). We all know that the screen has to be on to use apps, but the way your mobile app utilizes the other power-draining features of a mobile device can have huge effects on battery life.

Consumers have typically blamed their devices, the device manufacturers, or the carriers for device battery issues. This is no longer the case. In fact, 35% of consumers have uninstalled an app due to excessive battery drain.[13] Consumer tools that display how apps drain the battery are only just now coming to market, but the quality of these tools is radically improving. Thankfully, the tools for developers to minimize power drain are also beginning to surface, and we’ll explore these tools in Chapter 3. It is best to be as conscientious as possible regarding battery and power concerns while architecting and building your mobile apps.

Testing Your App for Performance Issues

The best way (prelaunch) to discover performance issues is to test, test, and test some more. In Chapter 2, I’ll cover the devices you should use for testing in order to cover as much of the Android ecosystem as possible. In subsequent chapters, I’ll walk through many of the tools available to help you diagnose performance issues, and tips to resolve them. Once you are in market, ensure that your app reports back to you on usage patterns and issues that your customers are facing. Read these reports, and dissect the information so that you can resolve issues discovered in the field.

There are two common areas of performance testing: synthetic and real user monitoring (RUM).

13 Hewlett-Packard, “Failing to Meet Mobile App User Expectations: A Mobile User Survey - Dimensional Research,” *http://bit.ly/1LXYPec*.

Synthetic Testing

Synthetic tests are created in the lab, to test specific use cases, or perhaps to mimic user behaviors in your mobile app. Many of the tools we'll discuss in future chapters run with synthetic tests, where you as a developer run your app through its paces and look for anomalies. This is a great way to discover and resolve many bugs and performance issues. However, given that Akamai reports 19,000 unique Android user agents per day,[14] there is no way you can possibly run a synthetic test for every possible scenario.

Real User Monitoring (RUM)

Testing every device scenario is impossible in the lab, so it is essential to collect real performance data from your customers. By inserting analytics libraries into your app, you can collect real-time data from all of your users—allowing you to quickly understand the types of issues they might be facing. This gives you the chance to respond to customer issues and bugs that are discovered in the field. Of course, once resolved, it is smart to find ways to replicate such issues in the lab—to avoid future releases with issues. Chapter 8 will walk through some typical results you obtain from RUM tools.

Conclusion

The evidence presented in this chapter conclusively shows that the performance of your app—load speeds, scrolling actions, and other user events—must be fast and smooth. A slowly performing app results in loss of customers at a rate similar to apps that crash. For that reason, having a poorly performing app is like operating with rolling outages. You'll lose engagement, sales, and ultimately your customers (both current and future). So, now that you are convinced (and once you've used this data to convince your managers and senior leadership, too!), let's go solve all your performance issues and get your app running fast!

14 Alec Heller, "UA Strings Are Terrible: Adventures in Server-side Device Characterization for Site Performance," Velocity 2014: Building a Fast and Resilient Business, June 25, 2014, *http://velocityconf.com/velocity2014/public/schedule/detail/35211*.

CHAPTER 2

Building an Android Device Lab

The Android ecosystem is the largest mobile platform (by market share) in the world. Google has reported that there are over 1 billion (yes, with a *B*!) active Android devices worldwide. It holds ~80% of all smartphone penetration. With these stats, it's no wonder that Android app development is hot. However, the rapid growth of the Android ecosystem has also introduced some pretty interesting challenges.

There have been 12 major releases, thousands of phone models (and tablets, watches, TVs, etc.), with dozens of screen sizes, all with manufacturer tweaks added to the standard Android Open Source Project software. With all of this variation, it is impossible to test your app on every device/OS combination. Akamai has reported that they track 19,000 unique Android user agents per day.[1] How can you make sure that your app is running well on a representative sample of Android devices? And probably just as importantly, how do you figure out what a *representative sample* of Android devices actually means?

A study by TestDroid found that to test the top 20% of devices globally, you need 12 devices. To cross 50% of devices, you need >60. For just the U.S. market, 25 devices covers ~66% market penetrations, but to hit 90% coverage, you need to actively test on 128 devices. As testing time is always at a premium, it is unlikely (without automation) that you will be regularly testing on this many devices. In this chapter, we'll walk through a few different options for building an Android device lab that will help you maximize, via functional and performance testing, your UI on a minimum of devices.

1 Alec Heller, "UA Strings Are Terrible: Adventures in Server-side Device Characterization for Site Performance," Velocity 2014: Building a Fast and Resilient Business, June 25, 2014, *http://velocityconf.com/velocity2014/public/schedule/detail/35211*.

What Devices Are Your Customers Using?

The easiest way to break down Android usage is by OS version, as your performance tuning will vary by the OS in use. As of June 2015, only 12.4% of Android users are running a version of Lollipop, while 39.2% are on KitKat (KK) and 37.4% are on Jelly Bean (JB) (meanwhile, 5.1% of Android users are still running Ice Cream Sandwich (ICS) and Gingerbread and Froyo combined still account for 5.9%). While new customers are flocking to the latest devices and latest OS versions, there is still a large audience on devices/OS versions launched over three years ago. The device breakdown will also vary depending on the region of the world, as high-end device sales will predominate in wealthy countries, whereas used devices (and low-end new devices) will predominate in developing countries.

Device Spec Breakdown

Android Gingerbread requires a device with a minimum of 128 MB RAM, and 2 GB of storage. We'll place that as the bottom tier of devices, and there are still many devices out there with this profile. Some devices have no camera, some have only a rear-facing camera, and some have both front- and rear-facing cameras. Start throwing in sensors like NFC, thermometer, accelerometer, and barometer, and you can see that there is a huge dichotomy in device specifications. Let's look at some of the most challenging specs that affect your development.

Screen

Screen size has always been a concern for Android developers, because if your app does not look good or render properly, your customers desert you. As I have mentioned, the huge disparity in screen sizes does not simplify this situation. Making sure that your app displays correctly on all devices is a crucial step in the dev process. In recent years, device screens in the United States seem to be getting larger and larger, with no regard for hand (or pocket) size. The Samsung Galaxy S (2010) was 122 mm in height and featured a pixel density of 480 x 800, whereas the S5 is 145 mm in height and has a pixel density of 1080 x 1920 (nearly an inch longer, and 5.4x the pixel density in just four years!). The latest Nexus 6 from Motorola maxes out the phablet category of devices with a massive 151 mm (that's 5.92 inches) screen height and 2560 x 1440 Quad HD resolution.

Despite this trend to larger and larger screens, there is still a sizable chunk of users with screens of 480 x 320 or smaller (according to a MarketingProfs study,[2] >5% in

---

2 MarketingProfs, "Mobile Trends: Most Popular Phones, Screen Sizes, and Resolutions," *http://www.marketingprofs.com/charts/2014/25740/mobile-trends-most-popular-phones-screen-sizes-and-resolutions#ixzz3hPrTffs4*.

www.allitebooks.com

Q2 2014), and 17% of users in South Africa are using phones 240 x 320, so while it might be tempting to only use big flashy screens in your test lab, keeping one or two small screen devices is a prudent choice. If budget does not allow, small screens might also be tested with emulators for UI work, but emulators are not great for measuring performance on real devices.

SDK Version

Devices on different SDK versions can have great variation in components and performance. Post-Jelly Bean devices benefit from "Project Butter," which helped make UI scrolling and rendering buttery smooth and avoid jank. The KitKat release included "Project Svelte," which reduced the memory requirements of the OS to allow devices with just 512 MB of RAM (and devices running KitKat with even just 256 MB have even entered the market). Lollipop introduced "Project Volta," with SDK improvements that save battery drain. While these updates are great, it also complicates your development for devices that fall before these releases.

While many devices are used for a short period (6–18 months) and then discarded, there are many other Android phones that see strong usage for over two years. The Samsung S3 (released in 2012) is still being sold as a new device in 2015 (alongside its successors, the S4, S5, and S6) and is one of the top 5 Android device in usage charts. There is also a very strong used Android market both domestically and abroad.

CPU/Memory and Storage

As recently as October 2014, the top two phones in India run Jelly Bean on a single core CPU with 512 MB of RAM and 4 GB of storage. In China, high-end devices similar to the ones popular in the United States and Europe are common, as are single-core devices running Gingerbread. Your app may run smoothly on common devices in the United States, but how does it react to a lower-powered device with less RAM or storage? As your app reaches the memory limits of the device, does it work to shed memory allocations, or does it just continue plugging ahead (slowing performance and likely leading to a crash)?

What Networks Are Your Customers Using?

We'll cover this in greater detail in Chapter 7, but in North America (especially in cities), we have ready access to high-speed LTE (the latest studies show 97% of the population has access to LTE or other 4G technologies). This drops to about 83% in Western Europe, and continues to drop even further in other parts of the world. Many areas have not even seen 3G, and are still served by older 2G networks. If you are planning a major international release of your mobile app, you should be ready to test your Android app on different network conditions. Techniques for emulating

slower networks will be covered in Chapter 7 so that you can ensure your app is running quickly no matter the location or network of your customers.

Your Devices Are Not Your Customers' Devices

The days of "only testing it on the phone in my pocket" are over. When I attend developer conferences, every attendee has a high-end phone—usually a flagship device from the last year or two. For 2015, that means developers are carrying devices equivalent to the Nexus 6, Samsung S6, or HTC One (M9), all of which run Lollipop and have 4 or 8 CPUs, multi-megapixel cameras, 16–32 GB of memory, 2–3 GB of RAM, and HD video recording capabilities. Many developers are also tinkerers, and have rooted their devices. These devices are awesome, and a lot of fun to use, but it is important to realize that these devices are not the norm for the general Android population. Further, Android developers tend to live in high-tech hubs, ensuring fast Wi-Fi and cellular networking capabilities.

With a billion active Android devices in use in extremely different network conditions, it is clear that developers live in a bubble of high-end devices on amazing high-end networks. Growth of mobile data continues to grow in the developed world, but it is beginning to reach saturation. The largest new user growth will be in the developing world, as the next billion users begin to gain access to the Internet. These users will be looking at low-cost devices, and there is a thriving market of Android devices that meet this need. These devices are markedly different from what is in your pocket (they likely resemble your retired device from a few years ago that you loaned to a family member and forgot about). These devices lack the horsepower we take for granted, but that's not the only consideration to keep in mind when catering to these users. We also have to realize the other limitations they may face—access to electricity to charge their phones, and the quality of the mobile network that supplies the data, for example.

Rooted Devices/Engineering/Developer Builds

Rooted devices are devices with root access to the Android kernel. Many developers are tinkerers and enjoy the access to the root Android kernel that rooting provides. As a developer, you should expect that your app will be run on rooted devices, and you should be prepared for any security issues that might arise from running on rooted devices (things like users accessing any file—even in your protected sandbox, meaning that anything sensitive cannot be stored on the device).

Rooted ROMs typically have a superuser Android application package (APK) installed as an interface between apps and the kernel (if you don't have one, there are several good options in Google Play).

Developer/engineering builds are a subset of root. The rooting community also calls these "insecure" builds. That is because debugging is turned on, and the security of the device is turned off. On an engineering build, you can access and debug any app on the device. This is a very powerful option, and a useful one for Android developers. On the downside, many Android apps have large security holes that are further exacerbated by root access. For testing, the ability to test with root access gives you more access to core levels of the Android OS. For the same reason, you should use a rooted device for personal use with discretion.

For some of the test tools we discuss later in the book, root access provides additional insights that can be useful. It may also be helpful for your security testing (which is out of the scope of his book), so I would recommend having one device with root access on hand for your testing.

Legal disclaimer: in some jurisdictions, rooting a device has legal/copyright implications. In the United States, it is currently legal to root Android phones (but not tablets), as long as there are no copyright infringements. If you are unclear on the legality in your jurisdiction, consult with legal counsel.

Testing

So, considering the challenges just described, how can we rationally break down the immense 1 billion user, ~20,000 device Android ecosystem to a manageable test bed of devices? How can we make sure we are equally supporting users in the developed world, while also ensuring the potential customers around the world are also being supported adequately? Hopefully by now, I've made it pretty clear that "testing with what's in my pocket" is not going to cut it.

In the next section, we'll discuss a few approaches toward building a device lab that covers your bases today and will keep you covered moving forward. Of course, every-

one's budget varies greatly, so feel free to add (but not subtract too much) from the device suggestions.

Building Your Device Lab

The only way to ensure that your Android app is doing what you want it to do is to test, and to test on as many screens and configurations as you can. For this, you need an Android device lab for testing.

Your device lab might just be a desk drawer of devices in various states of charging, tangled up in a mass of cables. I suppose the pro to this arrangement might be that devices are nearby, and easily secured when not at your desk. However, the cons probably outweigh the pros here: only you can access your "lab," the device you need is probably not charged, and perhaps most importantly, the "out of sight, out of mind" complex. If you are not constantly looking at your devices, you might forget about testing with them.

The alternative to a locked drawer of devices is an open access device lab. In this arrangement, devices are kept in a secure area, but are left out for people to easily access, sign out, and test with.

When acquiring devices for testing, it is crucial to ensure your device selection will complement the widest spectrum of your users, while sticking to your budget. If you already have an app in market, your analytics data can be very helpful to break down the devices that your customers are using (for more information on app analytics, jump to Chapter 8). Perhaps your users' device choices deviate from the top reported devices. To keep existing customers happy, you should ensure you always test on customers' top devices. If you don't have analytics specific to your app, you'll have to stick to reported data on top Android devices (however, these reports generally agree with one another, so the devices you choose from this data are a pretty safe bet).

You Want $X,000 for Devices?

Cost is always the elephant in the room. In this subsection, I'll walk through what an ideal Android device list will look like, but at the end, finances will come into play. You may be asked "why don't you just use the emulator for testing different devices?" The emulator can help you in a number of ways (having many different size emulator screens *might* help you with UI issues). But, as developers, we all know there are issues with the emulator (speed and inability to use sensors like location and accelerometer to name a couple). You'll have to convince your management that devices are essential for performance testing. Perhaps a meeting where you walk through the testing process with an emulator or three would be helpful (this actually worked according to a presentation made by the Twitter Android team).

Beyond budget, let's take a look at a variety of parameters or tests you may want to iterate over with different devices:

- Screen size
  - Small (4.4%)
  - Normal (82.9%)
  - "Phablet" (8.6%)
  - Tablet (4.1%)
  - Special cases (wearables, TVs, automobiles, etc.)
- Screen density
  - Low (4.8%)
  - Medium (16.1%)
  - High (40.2%)
  - Extra-High (36.6%)
- Processor
  - Dual core
  - Quad core
  - Octo core
- Memory
  - RAM
  - Storage (e.g., devices with nearly full memory versus devices with lots of space)
- Network speed
  - 3G
  - LTE
  - Wi-Fi
- SDK version
  - Gingerbread (2.3 versus 2.3.3 versus 2.3.7)
  - Ice Cream Sandwich
  - Jelly Bean (4.1 versus 4.3)
  - KitKat
  - Lollipop
  - Marshmallow (and beyond)

- Other considerations
  - Rooted
  - Security testing
  - Original equipment manufacturer (OEM) differences

The great thing about this list is that there is opportunity to mix and match these different characteristics to a (relatively) small number of phones. There are a number of ways to break down these many characteristics into discrete phone groupings.

So What Devices Should I Pick?

Assuming you don't have the fiscal (or time) budget to test 100 phones, let's figure out a methodology to pack as much device testing into as few devices as possible. There are a number of ways to source your devices, and none are better than the other. Consider the following examples:

- Facebook has gone an interesting route for its device testing.[3] Rather than source a selection of older handsets from Craigslist or eBay, they have chosen a group of current Android devices that have similar specifications to the top devices from each year back to 2008. This allows the Facebook team to emulate the user experience across the top phones of today, as well as popular phones from years past (that will also proxy for lower-end phones still being sold around the world today). In 2014, Facebook reported that the most common bucket of phone matched its "2011" phone class, a dual-core, 1 GB RAM device (Facebook uses the Samsung S2 to test this class of devices). They have released an open source library that identifies a device's "year class" so that (based on your profile of those devices) you can serve the appropriate content to customers using that specific device.
- Etsy uses its device analytics to discover which devices are popular, and did its initial sourcing from that list. Etsy sources used devices that do not have mint batteries so it's possible to test devices that have more realistic power drain for older handsets. As new devices are released, the Etsy team observes which devices are growing quickly among the site's users and adjusts its test devices accordingly.

A few other tips: if your app does a lot of heavy calculations, test on different CPU types. If you have a lot of heavy rendering, look for devices with large screens and smaller GPUs—as this might be a location of a performance bottleneck. In subsequent chapters, I also discuss how changes in the SDK have improved performance in

3 Facebook, Year class: A classification system for Android

different areas, so looking at older devices on earlier SDKs without those performance tweaks is also a good idea.

Popular yesterday

Devices that were top devices 24, 36, or 48 months ago make good references for "older devices." This might be a great device to pick up used on eBay or Craigslist, which will save money and accomplish getting an older SDK device with a smaller screen.

The Nexus S can be run on Gingerbread through Jelly Bean (but it has a relatively larger screen at 480 x 800), so in order to maximize your device portfolio, it might be better to choose an older device with a smaller screen—for example, a device like the Samsung Galaxy Y (with a 240 x 320 screen) to source your low-pixel density, small-screened Gingerbread device. This method will create some variation in your testing, as you can only source what is available to you at the time you are purchasing.

Popular today

Your analytics may paint a different picture, but as of 2014, several online sources show that the Samsung S3 (initially launched with ICS in 2012) is still the top used Android device. Additionally, the Samsung S2 (released in Q1 2011) remains in the top 10. These are great examples pointing to the staying power of popular devices. There are variants of ICS, JB, and KK for the S3 (the S2 was only upgradable to JB). While the S3 device share has plateaued (and, in fact, may be decreasing slightly), this device will certainly remain high in the established install base for a relatively long period. Adding current popular flagship devices is also a good idea, as they will serve your testing for years to come.

Popular tomorrow

Nexus devices (those sold by Google with a *pure* Google experience with no OEM modifications) are not typically subsidized by carriers, and so are not the highest selling or used devices in any user ranking studies. While that might cause you to not add these to your inventory, these are also the first devices to get OS updates. This will allow you to test your app on the latest OS releases before the mainstream devices are upgraded or launched with the new OS. With Android Marshmallow, Google began prereleasing the latest OS to all developers on the Nexus 5, 6, and 9, so keeping a recent Google Nexus device on hand is a good idea for *future-proofing* your app.

Beyond Phones

In addition to phones and tablets, Android is quickly morphing and migrating to additional ecosystems like wearables, TVs, and automobiles. These platforms are different from traditional Android devices, but depending on your development plans, you may want to integrate some of them into your regular testing.

Android wear

Announced at Google I/O 2014, Android Wear is a new breed of Android. Devices running Android Wear are typically Android smartwatches that communicate back to an Android device over Bluetooth (there is no unique phone number or SIM on the device). Google suggests different UI interactions with your watch that you would have with your phone. Information is delivered in a series of cards that users can interact with. Google breaks down interactions to:

Suggestions
: A timely list of information for the user (e.g., messages, location relevant data, etc.)

Demands
: Allowing voice commands to control your Wear to ask for data

This development model is significantly different from traditional Android apps, so if you plan on building apps for Android Wear, you should have one or two representative devices in your lab.

Android Open Source Project Devices

Something that is often missed in the United States when discussing Android is that it's open source, and the Google version we are accustomed to use is simply one fork of the Android Open Source Project (AOSP). In the preceding description of Android devices, I have purposely focused completely on the various Google incarnations of Android, as they are the predominant devices in the United States. However, in an effort to be complete in my coverage of Android, let's take a look at other common forks of the AOSP.

As recently as summer 2014, studies have estimated that AOSP devices are 20% of the smartphone market (where Google's fork accounts for 65%). These devices differ as they do not have:

- Google Play Store for app distribution
- Google Cloud messenger push notification
- Google Play Services

- Google products and apps, among other tools that have been customized by Google

However, as this ecosystem is not insignificant, you should consider these devices as a part of your app distribution strategy.

Amazon

The most common device in the United States running a fork of Android is Amazon's popular e-reader, the Kindle. Amazon has also launched into phones (the Amazon Fire Phone), and TV set-top boxes (Fire TV). Amazon calls its fork of Android "Fire OS," and its variants correspond to the Android SDK versions:

Fire OS 1	Gingerbread	2011
Fire OS 2	Ice Cream Sandwich	2012
Fire OS 3	Jelly Bean (4.2.2)	2013
Fire OS 4	KitKat (4.4.2)	2014
Fire OS 5	Lollipop	2015

It might be useful to have a few Amazon devices in your lab, as Amazon does have its own unique App Store to deliver content to all of the Fire tablets—this is another market for your Android apps. As long as you do not use Google's specific services, adding your app to Amazon's ecosystem (including the Amazon website) is a smart idea. Android apps available in the Amazon App Store are also available on BlackBerry devices, which have a runtime that allows for Android apps to run on them.

Other Android phones/tablets

While the most popular non-Google Android devices are the Amazon devices, other devices fitting this category found in the Unites States include Barnes & Noble's Nook tablet. Nokia had a short-lived Nokia X AOSP project before being acquired by Microsoft.

Outside of the United States, there are a number of manufacturers successfully marketing AOSP devices. These are primarily OEMs in India and China, where delivering inexpensive phones is tantamount. For example, the Chinese OEM Xiaomi holds 5.1% global marketshare, and had one billion app store downloads in just over a year with their MIUI fork of Android. If your target market includes *the next billion* connected users (hint: it probably should), these devices should be considered for testing.

Other Options

If sourcing and maintaining a lab of devices is not possible, there are other options available for testing devices.

Remote Device Testing

There are services online that provide you access to real devices connected via various web interfaces. Testdroid, Appurify, Perfecto Mobile, and Keynote are among the leading vendors that have mobile devices online available for testing. These services take care of the device overhead, allowing you the ability to just test your apps. These services typically have many top phones, and allow you to run scripts, or other continuous integration processes to test your apps. The results can then be viewed in your browser. These services are unlikely to save you money on testing costs (in fact, it will likely cost you more), but they do remove the headache of maintaining devices locally. Another disadvantage is that without actually handling the physical devices, you are unlikely to see slowness or performance issues—you'll instead focus mostly on the test results, and not actually see your mobile app in action on these devices.

> **Google's Cloud Test Lab**
>
> At Google I/O 2015, Google announced a new service of online physical devices "nearly every brand, model and version of physical devices your users might be using, to an unlimited supply of virtual devices in every language, orientation and network condition around the world." Submitted APKs will automatically be tested across the top 20 Android devices for free—reporting results and crash data. As of summer 2015, this tool is still coming soon.

Open Device Labs

If your budget for devices is truly zero, or perhaps you are afflicted with a bug on a device you just cannot manage to get your hands on, you might try an Open Device Lab (ODL). These are grassroots device labs (some have permanent homes, some do not) of devices that are available for testing (see Figure 2-1). The number of devices for these labs vary, but perhaps you can find some old devices to donate to your area's ODL. If your community does not have an ODL, perhaps you can start one. All you really need is a few old devices, and a willingness to share good testing karma with your fellow developers.

www.allitebooks.com

Figure 2-1. Open Device Lab locations around the world

Additional Considerations

When building a device lab, there is additional infrastructure you must acquire and maintain. Lara Hogan at Etsy has done a great job of discussing device lab issues beyond just device acquisition.[4] Other things to consider include:

- Obtaining USB hubs to ensure you have enough electricity to power all of your devices
- Setting up a private Wi-Fi network just for your mobile devices (to ensure adequate Wi-Fi throughput)
- Ensuring all the devices are wiped after each use, and that they're not accidentally upgraded
- Having the appropriate cables and chargers for each device.

These additional details are crucial to getting your device lab up and ready for your developers to begin testing. Software that controls your mobile devices can also be used to run basic synthetic tests on your device lab.

4 Lara Hogan, "Etsy's Device Lab," Etsy - Code as Craft, August 9, 2013, *https://codeascraft.com/2013/08/09/mobile-device-lab/*.

My Device Lab

In 2015, my family and I are taking an extended trip to Europe. I will be working on this trip, so I've put together a portable device lab that I will carry for the entire trip. For my work, I am not typically worried about screen sizes, but I am interested in how apps work on popular older versions of the Android OS. I also often test with rooted devices. With that in mind, I will be carrying the devices shown in Table 2-1.

Table 2-1. Portable device lab

Device	Operating System	Year
Samsung Galaxy Note II	Jelly Bean (rooted)	2012
Samsung Galaxy S4	KitKat (rooted)	2013
Motorola's Moto G	Lollipop (rooted)	2013
Nexus 7	Lollipop	2013
Moto X(2014)	Lollipop	2014
Nexus 6	Marshmallow	2014
HTC One M9	Lollipop	2015
Samsung Galaxy Note 5	Lollipop	2015

This list is heavily weighed on the Lollipop side, but I expect to upgrade most of these devices to Marshmallow when the update becomes available. This list gives me a good selection of OSes, a three year range on device releases, and OS versions that cover over 85% of the population—and is additionally future-proofed for the near term.

Conclusion

There are a multitude of Android devices, and there is growth into new sectors of entertainment and travel. There is no reason to believe that this is the end of Android's growth; think of all the things that might run Android in your home—for example, controlling your Android coffeemaker from bed.

Making sure that your app runs well on phones, tablets, cars, TVs, watches, sunglasses, and more requires a dedicated effort, and (to the chagrin of some) lots of testing. By obtaining a reference library of devices (and setting them up so that you can test with them easily), you will be on your way to optimizing the performance of your app, and thus making your customers happier, which will help you grow your audience. In the subsequent chapters, we'll look at how to test your app on these devices to ensure that the performance is optimal for every user on every phone.

CHAPTER 3

Hardware Performance and Battery Life

In addition to fancy cameras, bigger and brighter screens, and faster, more compact processors, one of the biggest technical features touted with new devices is the size of the battery. Reviews of new devices chart how long the device's battery will last compared to previous generation models. As users of smartphones, we have taken to carrying battery chargers with us—we have chargers at home, work, and in our cars—to make sure that our devices remain powered.

It is my contention that the devices are fine. The problem is that after you walk out of the store with your new phone, you begin to install apps. Yahoo! reported in 2014 that the average Android device has 95 apps installed, but only 35 are used daily.[1] As these apps begin running, they utilize the various hardware functions of the device, and battery drain begins. As customers become more cognizant of this fact, we'll see more tools to help consumers find apps that are causing high amounts of battery drain.

In this chapter, we'll look at how apps can utilize the device hardware, and how important it is to optimize these interactions—to speed up the performance of your app. Additionally, by improving the way your app interacts with the device, you will reduce your app's impact on battery life.

Android Hardware Features

With the number of sensors on today's Android devices, it seems that there is nothing you cannot do with them. However, as Uncle Ben told young Peter Parker (the fledgling Spider-Man) "With great power, comes great responsibility." Android devices

1 Paul Sawers, "Android Users Have an Average of 95 Apps Installed on Their Phones, According to Yahoo Aviate Data," The Next Web, August 26, 2014, *http://bit.ly/1JvbPGk*.

provide developers with a lot of cool tools, but like any carpenter will tell you, you have to be careful with your tools or you might hurt yourself. In this case, you won't physically get hurt, but if your app does not work in concert with the device, you can cause major battery drain, device warming, and other negative side effects that might alarm your customers.

The power we hold in our hands is pretty amazing—for example, consider the sensors included in the Samsung S5:

- Fingerprint scanner
- Heart rate monitor
- Light monitor
- Relative humidity and temperature sensor
- Barometer
- NFC
- Gyroscope
- Accelerometer
- Bluetooth
- Wi-Fi
- FM radio
- Cellular radio
- Front- and back-facing camera
- GPS
- Magnetic field detector
- Light flux
- Battery temperature sensor
- Microphone
- Touch capabilities

How do we quickly understand the performance aspects of all of these sensors? The easiest way is to look at power drain. The parts of the device that consume the most power are also those you need to be most careful with.

Less Is More

By utilizing these awesome features of our customers' Android devices, we want to gain as much information as possible, and provide it to our customers. The challenge is that if we collect too much data, we impact the battery life of the device, and so we

must discover the correct balancing point of acceptable data/information with power consumption. Further, if we can ensure that all tasks run as quickly as possible, we can be sure that the performance/battery pendulum is swinging in our favor.

Google has reported that 1 second of active device usage is equal to the power drain of 2 minutes of standby time. This makes sense for anyone who has looked at device specs. The Nexus 5 boasts 300 hours (12.5 days) of standby time (which means LTE on and Wi-Fi on, but no device usage.) As soon as customers begin installing apps (or turning on the screen to check on said apps), battery life drops a whopping 35x! (The Nexus 5 promises 8.5 hours of battery life with regular Wi-Fi usage.) Looking at the bigger picture, we can assume that 5 minutes of active app usage will draw 1%–1.6% of the battery. The more *stuff* your app uses, the higher this number will be.

This is most evident on free, ad-supported casual Android games—sometimes, after playing these games for 10–15 minutes, you discover that the back of your phone is hot to the touch. These apps can aggressively download advertising while the game is using the CPU, screen, and so on. All of these components working at once drains the battery so quickly that it heats up. A study released in March 2015 found that apps with ads used 56% more CPU, 22% more memory, and 15% more battery over the same app with ads stripped out.[2]

It is my contention that most battery issues with mobile devices are not hardware related, but rather, the result of poorly designed apps that misuse the capabilities of the device. In this chapter, we'll walk through some of the missteps of hardware usage, and how to avoid them in your Android app (and thus stay off any "battery drain" app list).

What Causes Battery Drain

As an Android user, you are probably interested in how the apps *you* use on a regular basis affect your battery life. By studying the apps currently installed on your phone, you may discover apps that are using excessive battery. By learning these techniques, you'll be able to determine if your customers will discover similar performance issues with *your* app on their phone. By understanding how Android grades apps for battery drain, you can ensure that your apps do not appear on these reports. You may also discover some poorly behaving apps on your phone (thereby improving the battery life on your personal device).

2 Jiaping Gui , Stuart Mcilroy , Meiyappan Nagappan, and William G. J. Halfond, "Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers," Proceedings of the 37th International Conference on Software Engineering (ICSE), May 2015, *http://www-bcf.usc.edu/~halfond/papers/gui15icse.pdf*.

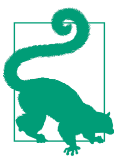
Android Power Profile

As we'll discuss later in the chapter, the battery settings menu reports the percentage of battery drain for each app running on the device. These power drain calculations are created (in part) by the Android Power profile. Inside the Android OS is an XML file that tells the system the electrical current drawn by the major hardware components of your device. When your app runs (and wakes up different parts of the device), the system computes the amount of power each component uses, and assigns that battery drain to your processes. The XML file looks like this (power drain reported in mA):

```
<?xml version="1.0" encoding="utf-8"?>
<device name="Android">
    <item name="none">0</item>
    <item name="screen.on">65</item>
    <item name="screen.full">202</item>
    <item name="bluetooth.active">87</item>
    <item name="bluetooth.on">1</item>
    <item name="wifi.on">3</item>
    <item name="wifi.active">240</item>
    <item name="wifi.scan">129</item>
    <item name="dsp.audio">29</item>
    <item name="dsp.video">215</item>
    <item name="radio.active">125</item>
    <item name="radio.scanning">25</item>
    <item name="gps.on">1</item>
    <array name="radio.on">
        <value>4.5</value>
        <value>4.5</value>
    </array>
    <array name="cpu.speeds">
        <value>2457600</value>
        <value>2265600</value>
        <value>1958400</value>
        <value>1728000</value>
        <value>1574400</value>
        <value>1497600</value>
        <value>1267200</value>
        <value>1190400</value>
        <value>1036800</value>
        <value>960000</value>
        <value>883200</value>
        <value>729600</value>
        <value>652800</value>
        <value>422400</value>
        <value>300000</value>
    </array>
    <item name="cpu.idle">3.1</item>
    <array name="cpu.active">
        <value>348</value>
        <value>313</value>
```

```
        <value>265</value>
        <value>232</value>
        <value>213</value>
        <value>203</value>
        <value>176</value>
        <value>132</value>
        <value>122</value>
        <value>114</value>
        <value>97</value>
        <value>92</value>
        <value>84</value>
        <value>74</value>
        <value>56</value>
    </array>
    <item name="battery.capacity">2800</item>
    <array name="wifi.batchedscan">
        <value>.0002</value>
        <value>.002</value>
        <value>.02</value>
        <value>.2</value>
        <value>2</value>
    </array>
</device>
```

The hardware with the highest power drains on today's mobile devices are (not surprisingly) the screen, radios (cellular, Wi-Fi, Bluetooth, and GPS), and the CPU (at high processing rates). As we look to optimize app performance, the same components that affect performance also affect device battery drain. So, by optimizing the performance of your app, you'll also be improving the battery life of your users' devices.



Power Profile

The Power Profile XML is found inside an APK that is part of the Android system. From the File Explorer in Android Monitor, browse to */System/Frameworks*, and copy the *frameworks-res.apk* file to your local machine. You'll need to decompile the APK to extract the *res/xml/power_profile.xml* file.

All of the values in the Power Profile are reported in milliamps (mA). As you might recall from your high school physics class, mA is a measure of current—or flow of charge. The higher the value, the faster the feature will drain the battery. The battery capacity is reported in milliamp-hours (mAh), or the amount of current that flows in one hour.

Screen

As seen in the Power Profile, the screen is one of the top causes of battery drain (when the brightness is set to a high value, the current drain approaches `screen.full` 202 mA in the power profile). As the screen is the essential UI element of your Android app, and it typically needs to remain lit while your app is running, you may feel that you have little control over this aspect of power consumption. However, there are certain UI aspects you can utilize to limit the battery drain from screen usage.

In general, there are two major screen types in Android devices: light-emitting diode (LED) and liquid crystal display (LCD). Manufacturers have proprietary versions of these screens—for example, Active Matrix Organic LED (AMOLED) or Super LCD3—and these will have different view and power aspects. However, at the high-level analysis we are looking at here, we can stick to just two screen types.

LCD

In simple terms, LCD screens consist of thousands of liquid crystals that generate the color for each pixel, and a backlight that illuminates all of them at the same time. Creating the color for each pixel takes minimal energy. The major energy cost for this type of display is the light that shines through the liquid crystals. This means that in general terms, each pixel costs the same amount of energy, no matter the color shown.

LED

For LED screens, each pixel emits both the color and the light. Each pixel is created by an arrangement of red, blue, and green LEDs (and these arrangements are highly sophisticated and display enthusiasts have contentious debates on which are the best). By modifying the brightness and color of each LED, the pixel can take on the desired color. Because each pixel is represented by three light sources, with intensity slightly different depending on color, the amount of power used for different colors is variable, depending on the color displayed. Black, being the absence of all colors, uses zero power, whereas white, being all three colors mixed at high brightness, will use higher power. In general terms, this means that darker colors will use less power than lighter colors. This is one major reason why some news and and social media apps (apps that contain lots of blank screen) use a black background.

There is minimal win for black backgrounds in LCD screens, but the potential power gains from LED screens are big enough to consider dark backgrounds for screens that are kept open for long periods of time.

www.allitebooks.com

In Chapter 4 we will cover screen performance and UI performance in greater depth —beyond simple battery and power analyses.

Radios

As Power Profile shows, cellular and Wi-Fi radios use similar amounts of power. In Chapter 7, we'll look at the differences in connectivity between Wi-Fi and cellular. In general, cellular connections are kept on for a longer time than W-Fi, making the cellular radio sessions longer and ultimately using more battery than connections made on Wi-Fi. At a high level, the best way to improve your app's use of the radio is to download as much as possible all at once and turn the radio off when you are done. This has a twofold improvement to performance. By reducing the number of requests, the screen can load faster, and you reduce the battery drain (but we'll cover this in greater detail in Chapter 7).

Another (sometimes forgotten) radio receiver is the GPS. When using location, knowing the accuracy of the positioning you need can save a lot of time (and power). If you only need a course location, the cellular network can often provide enough data that the GPS radio might never turn on. Because the tower location is stored on the device, if your customer is not moving, this might not even require a cellular radio connection. By avoiding the use of GPS, your app will be faster (the location is available on the device), and will use less power.

GPS Failover

Don't forget to account for the fact that your user might be inside a concrete bunker, and GPS satellites may not be visible to the device. If you don't get a GPS fix in a *reasonable* amount of time, make sure you turn the GPS off.

It is not uncommon to see Android apps that fail to do this, and keep the GPS on for 40 minutes without getting a location. This behavior was not good for the device battery, nor did it provide an improved experience to the customer.

In Chapter 7, we'll detail network performance in greater detail.

CPU

If your app is a game or has a lot of heavy calculations, you know you will be hammering the CPU hard. Additionally, if your app requires background calculations to occur, the CPU might be woken up to do additional computations. As the XML power profile shows, the higher the CPU is running, the higher the battery drain.

CPU usage is influenced by the screen, network, and all of the calculations that occur in your device. We will cover opportunities to optimize CPU usage throughout the book.

Additional Sensors

The Power Profile lists the major components of an Android device. Additionally, as we discussed in Chapter 1, our phones have many additional sensors that allow us as developers to really make our apps shine.

When you register a sensor, you can use the `getPower()` method to obtain the power drain of the sensor. As you might expect, there are a number of free apps in Google Play that list all of the sensors on a device and their associated current usage (in milliamps). For the Nexus 6, I find:

Accelerometer	0.3mA
Magnetometer	10mA
Gyroscope	1.5 mA
Proximity	12.675 mA
Light	0.175mA
Barometer	0.004 mA
Rotation Vector	11.8 mA
Geomagnetic Rotation Vector	10.3 mA
Orientation	11.8 mA
Linear Acceleration	1.8 mA
Gravity	1.8 mA
Tilt	0.3 mA
Device Position Classifier	5.6 e-45 mA
Step Detector	0.3 mA

Each sensor can report events up to a certain maximum frequency. As a developer, it is important to use sample rates that make sense for your app. In addition to the sensor, the CPU and memory of the device are used to handle the data and oversampling wastes these resources. There are a number of sample rates built into Android (`SENSOR_DELAY_NORMAL`, `SENSOR_DELAY_GAME`, etc.) that allow you to use industry standard sample rates.

Finally, when you are done using the sensor, make sure you unregister it. If you keep your listener active, the sensors will continue to report data, and this will lead to unneeded processor load, memory usage, and battery drain.

Heisenberg's Uncertainty Principle

Quantum mechanics in a Android book? Werner Heisenberg's research told him that by observing the world, we inevitably disturb it. When you run tests on a device to monitor battery usage, your test is also using battery—and slightly perturbing your results. There are many external tools that you can use to connect to a device that will not affect the battery drain during measurements. I have used the Monsoon Power meter (and several of the tools in this book allow you to integrate reports from Monsoon into them).

Get to Sleep!

Letting your app go to sleep when it is not doing anything is important. Releasing the sensors and radios and allowing the screen to turn off will go a long way to saving battery power. While letting your app go to sleep is crucial, it is also important to carefully evaluate how your app wakes up. By being mindful of how often your app wakes up the device, you will go a long way to saving your customers' battery life.

Wakelocks and Alarms

Historically, developers have used wakelocks and alarms to wake up a device to process information. Because it is likely that you might want your app to wake up and process some data without customer interaction, you are probably utilizing an alarm or wakelock today in your app. Wakelocks can also be used to prevent the device from going back to sleep. Now that we have looked at how much power each piece of Android's hardware uses, you will begin to see how waking up the device in the background can be detrimental to battery performance of your app and the device. Additionally, when your app wakes up a device, it opens the door for other apps to process events, perhaps turn on the radio. In Lollipop, Android added the JobScheduler API, which allows for smarter and synced device wakeups (we'll cover this API in more depth in "JobScheduler" on page 66).

Wakelocks

Wakelocks give you the ability to wake up (or keep awake) parts of the mobile device. When used properly, this is a useful feature for apps. I remember a car racing app that did not use a screen wakelock, and the screen would turn off mid-race and I would crash. Needless to say, I uninstalled that game! A screen wakelock is how movie streaming apps keep the screen from timing out during a movie, or a streaming music app keeps the audio channel playing while the rest of the device is asleep. In certain types of apps, these wakelocks are paramount to the user experience. However, if not handled properly, wakelocks can also cause extreme battery drain.

If only to emphasize this case, the wakelock is a part of the PowerManager API. The first paragraph of this class description reads:

> This class gives you control of the power state of the device. *Device battery life will be significantly affected by the use of this API.* Do not acquire `PowerManager.WakeLocks` unless you really need them, use the minimum levels possible, and be sure to release them as soon as possible [emphasis added by Android].

You'll notice that this advice is similar the advice given by Android for sensors, as your wakelock is keeping the device awake. As soon as you can let the device sleep, make sure you release the wakelock.

Wakelock Detection

If you are testing on a pre-KitKat device, there are a number of free wakelock detection apps in Google Play that are useful for diagnosing wakelock issues. These apps all generally do the same analysis, so pick one to test your app's wakelock usage. In KitKat and later, the wakelock detection APIs became system stats (a part of `adb shell dumpsys batterystats`), and are only available to these apps if your phone is rooted. We will look at the tools build into `batterystats` and how you can analyze your app using these tools.

Alarms

Alarms allow you to set the time that specific operations will be run. These are typically run when your app is not in the foreground, and often when the device is asleep. For example, "wake up the device every 60 minutes and check in with the server for updates." While this is one way to update your app, it can have side effects. As the Android SDK warns: "A poorly designed alarm can cause battery drain and put a significant load on servers."

I have encountered popular apps that use alarms to sync data. For example, one particular app turns on its customers' phone and establishes a cellular radio connection every 3 minutes to poll for news updates. These 480 *extra* connections per day

(assuming the phone battery lasted 24 hours), caused 10%–20% battery drain—just in the background.

When using alarms, you should only call an exact alarm if you need to alert at a precise time (like if you are building an alarm clock app). Otherwise, you can use an inexact alarm where the OS will coordinate all of the alarms to minimize battery drain. The following example will wake up the device once a day, at approximately `alarmTime` (meaning that the OS will coordinate the wakeup, but they won't be precisely 24 hours apart):

```
alarmManager.ssetInexactRepeating(AlarmManager.RTC_WAKEUP,alarmTime,
        AlarmManager.INTERVAL_DAY, alarmIntent);
```

Doze Framework

As we have seen in this chapter, the more the device is woken up, the faster the battery will be drained out. When the device is idle, the wakelocks and alarms that each app uses will accelerate the drain. Studies have shown that 70% of battery drain when the device is idle is caused by apps turning on a radio connection to update. We'll look at optimization strategies for network connectivity in Chapter 7, but it is safe to say that limiting the number of times your app wakes up will go a long way to saving battery.

In the 2015 Marshmallow release of Android, Google has added a Doze framework to limit how often a device can wake up. This comes at a price of "data freshness" in the apps, but having fresh data in your apps means nothing if the battery dies. The device allows certain windows to update (and of course when the device screen is powered on, all apps can update).

So how does the Doze framework work? The framework has several states:

`ACTIVE`
: Screen is on

`INACTIVE`
: Screen is off, but device is awake

`IDLE_PENDING`
: "Nodding off" into Doze

`IDLE`
: Device is asleep

`IDLE_MAINTENANCE`
: A short window for all queued alarms and updates to occur

To force a device running Marshmallow into these different states:

```
adb shell dumpsys battery unplug //tricks the device to stop charging
adb shell dumpsys deviceidle step
   //reusing this step walks you through the various states
```

In real life, your device must have the screen off, and not move for 30 minutes to go from `INACTIVE` to `IDLE_PENDING`, and another 30 minutes to go into `IDLE` mode. Once in `IDLE`, the device will postpone all alarms until the next maintenance window (in 60 minutes.) The delay between each `IDLE_MAINTENANCE` increases (1 hr, 2 hr, 4 hr, and 6 hrs) with a maximum of 6 hours between windows. All alarms and wakelocks will be suspended until the window occurs. This will undoubtedly save significant battery for devices that sit idle for long periods (like tablets).

As a developer, you should test your app with the Doze framework to ensure that if multiple notifications occur while the device is Dozing that only one alert message/tone is made.

Basic Battery Drain Analysis

We've covered how the hardware uses the battery, how Android calculates app battery drain from these values, and how inefficiently waking up your app can cause large battery drain. If apps are the cause of battery drain, how can you determine what the top battery drainers are on your device? The battery settings menu has a wealth of information to diagnose battery drain issues stemming from mobile apps, and more importantly, is accessible to all Android users. It is imperative that your app not appear as a battery hog in these menus, as it is a great tip-off to your customers to uninstall your app.

As shown in Figure 3-1, when you initially open the Battery menu (Settings → Battery), you can see a general graph of battery drain over time (typically since the last 100% charge). Below the graph is a list of all of the apps that have contributed to battery drain over the selected period. Let's walk through what these graphs tell you (and your customers).

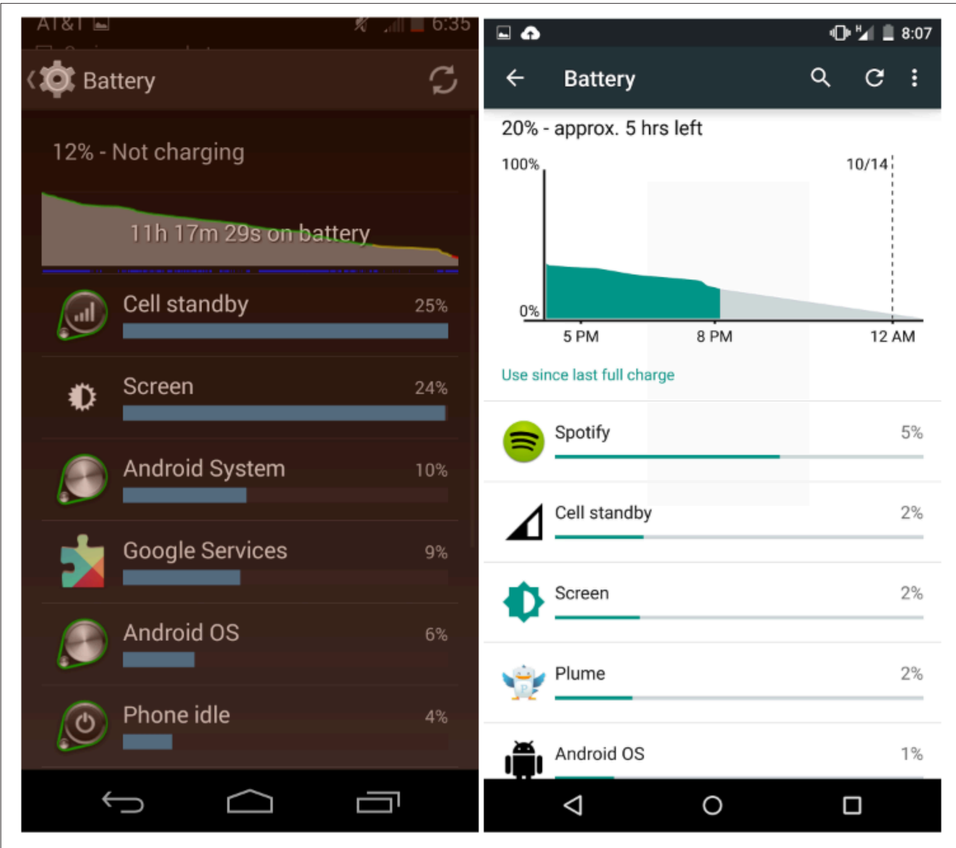


Figure 3-1. The KitKat (left) and Lollipop (right) battery menus

You'll see that the menu been updated between KitKat and Lollipop. The KitKat menu shows current battery usage, while Lollipop shows both usage and additionally predicts the battery life remaining until you must recharge (based on your usage). By touching the graph at the top, it will expand to show more device-specific details about what the device has been doing (Figure 3-2).

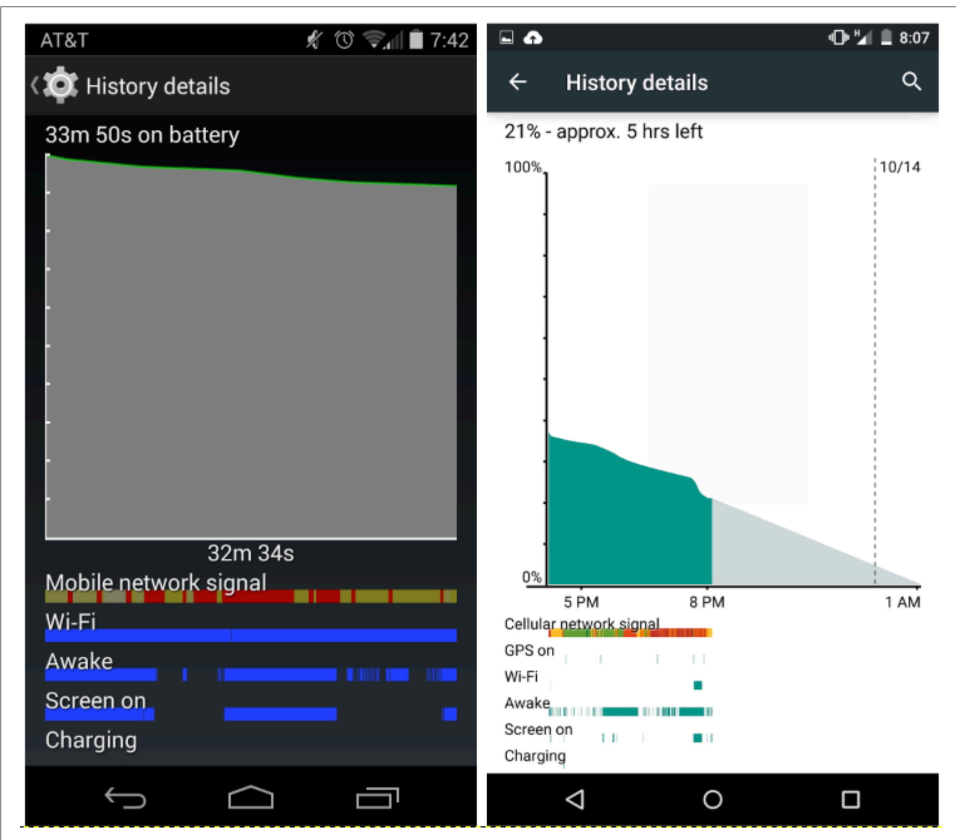


Figure 3-2. KitKat (left) and Lollipop (right) battery details

This extended menu tells you how much time your device has spent in various cellular states (green/yellow/red indicating the signal quality), time on active Wi-Fi, device awake time, screen awake, and how much time your device was charging. As a user, I prefer the Lollipop view, as it shows both the battery actual data (green), but predicts the time remaining (gray). As a developer looking at device and app performance, I prefer the KitKat view, because the actual device usage fills the screen, making it easier to read.

In the KitKat image, you can see the rapid discharge of battery (at the left of the screen) occurred during a period of poor signal, while the screen was on and the phone was awake. There is a similar dip on the Lollipop device just before the screenshot was taken (where the graph goes from green to gray.)

As a developer (and as a user), an important indicator of app-induced battery drain to note is when the device is awake but the screen is off. This is an indicator of an app using a wakelock or alarm to use the device while the customer is not using it. If you

see this occurring frequently, you can look at the apps causing battery drain, and determine which app(s) are causing the issue.

App-Specific Battery Drain

If you return to the main battery menu screen and scroll below the battery chart data, there is a breakdown of every app associated with battery drain. In my experience, the percentages are never very large, but this might be because every app is responsible for a very small percentage of battery drain. By selecting a specific app from the menu, you'll see the CPU usage of your app in the foreground and total CPU usage (a sum of foreground and background). Additionally, this menu provides data usage (foreground/background cellular/Wi-Fi), and the time the app kept your device awake. Foreground app usage and data are great (yay! people are using your app), but a large amount of background usage implies that your app might be waking up the device from its asleep state.

For example, Figure 3-3 shows the battery stats for Facebook and Spotify (in KitKat).

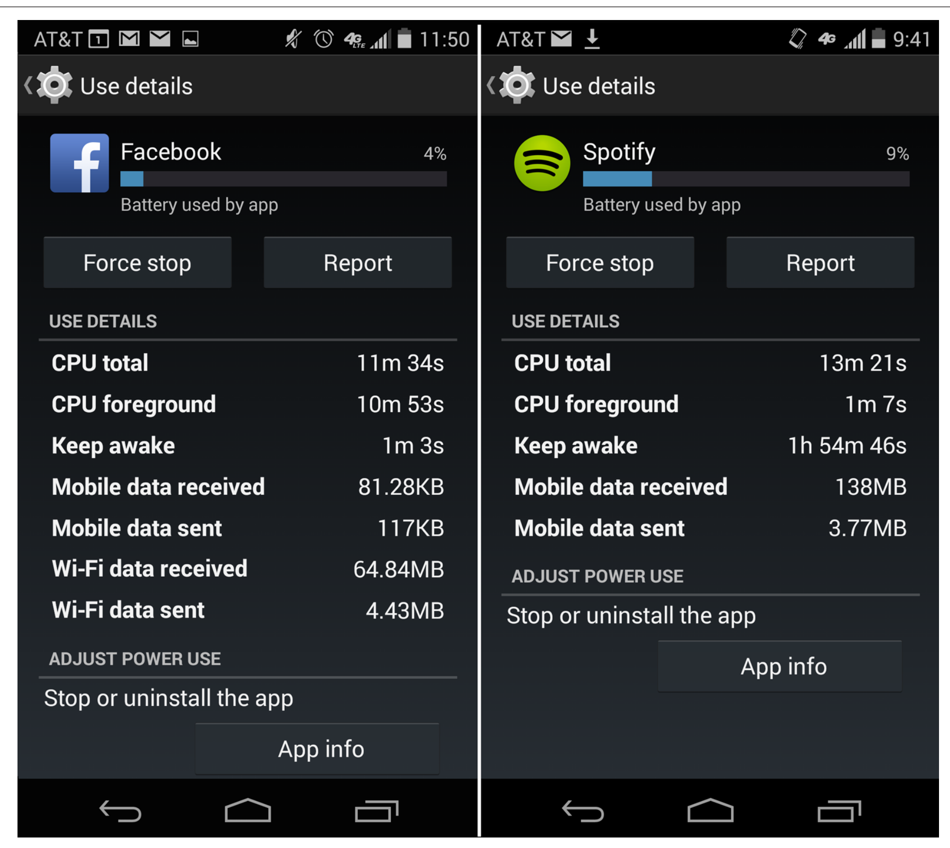


Figure 3-3. Facebook and Spotify battery details (shown in KitKat)

In this view, Facebook is being credited with (blamed for?) 4% of the battery drain (calculated in "Android Power Profile" on page 30) on the device. Facebook's CPU usage is primarily in the foreground (11 minutes out of 11.5 minutes). The additional 30 seconds of CPU usage was in the background, and probably associated with downloading updates from the server. The app kept the phone awake for 1 minute with a screen wakelock. This is because I watched a 1-minute video on my newsfeed, and the wakelock kept the screen from turning off. Facebook did not use a large amount of cellular data, but the Wi-Fi data totals are pretty impressive (but not unexpected: in addition to the movie, there were a large number of images downloaded).

Spotify usage is markedly different from Facebook. When streaming music, my screen was mostly off (and my phone was stashed in a pocket). The battery chart corroborates this; most of the CPU processing occurs in the background (~12 minutes) and the device is kept awake (likely with an audio wakelock) for almost 2 hours. The

www.allitebooks.com

data traffic is high, but not excessively so for 2 hours of streaming music (but without experience looking at these apps, it would be hard to know this).

The data usage information in the battery menu is the amount of data sent and received since the last charge (when the battery stats reset automatically). Reporting the data tonnage since the last charge does not tell you much about the efficiency of that data transmission (which, in my opinion, is what you really want to see in a battery menu). Tellingly, Google changed the reporting in Lollipop on the battery menu with respect to data usage, as shown in Figure 3-4.

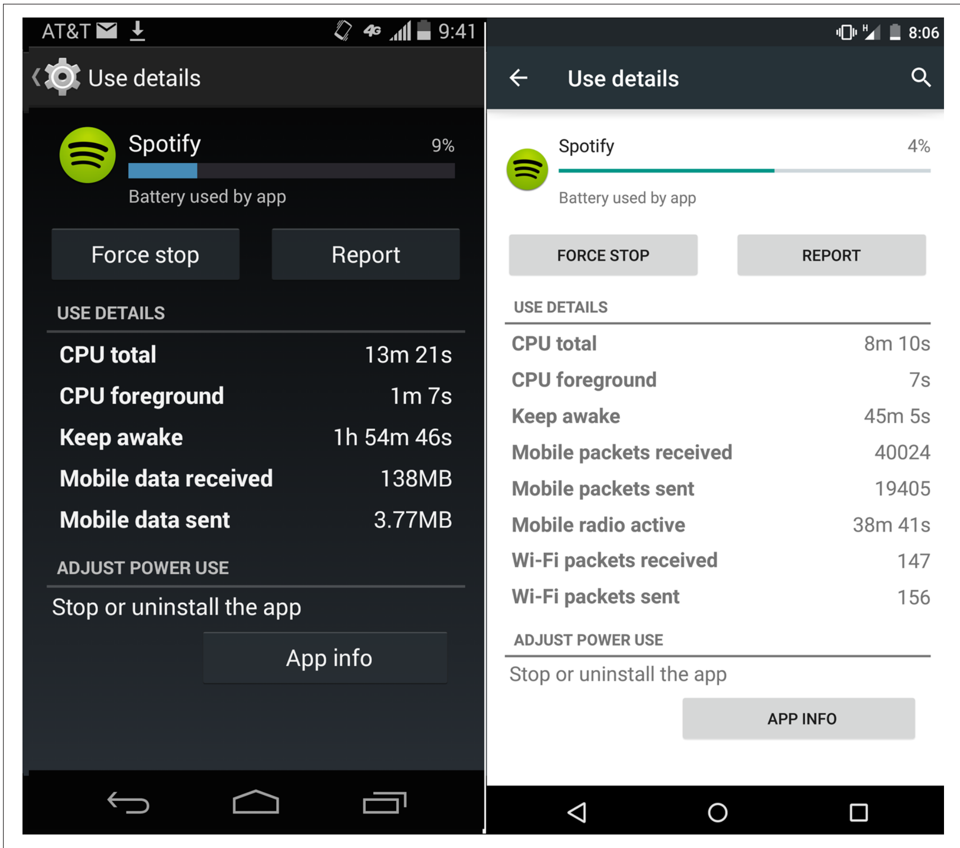


Figure 3-4. Battery details for Spotify, shown in KitKat (left) and Lollipop (right)

Note that the data usage report in the Lollipop battery menu is now based on packets instead of KB, and is further broken down into Wi-Fi and cellular categories. Further, the amount of time the mobile radio was in use by the app is also reported. These are powerful new reports, as we can now estimate how dense the radio traffic is (and as Spotify is sending 40,000 packets in 39 minutes, or 1 packet received every 60 ms, it is pretty dense traffic). Dense radio traffic implies that the data is being sent as quickly as possible, allowing the user to consume the data, while also minimizing the amount of time the radio is on.

Coupling Battery Data with Data Usage

To get a better handle on data usage of mobile apps, you can use the Data Usage menu (note this is cellular only—no Wi-Fi traffic is recorded here, as Wi-Fi is typically unmetered). When you select an app, you are provided with the amount of data used in the foreground and background. In the screenshot shown in Figure 3-5, I have moved the sliders to show only the data for 2 days, allowing me to pinpoint foreground and background data usage for Facebook for just 24 hours.

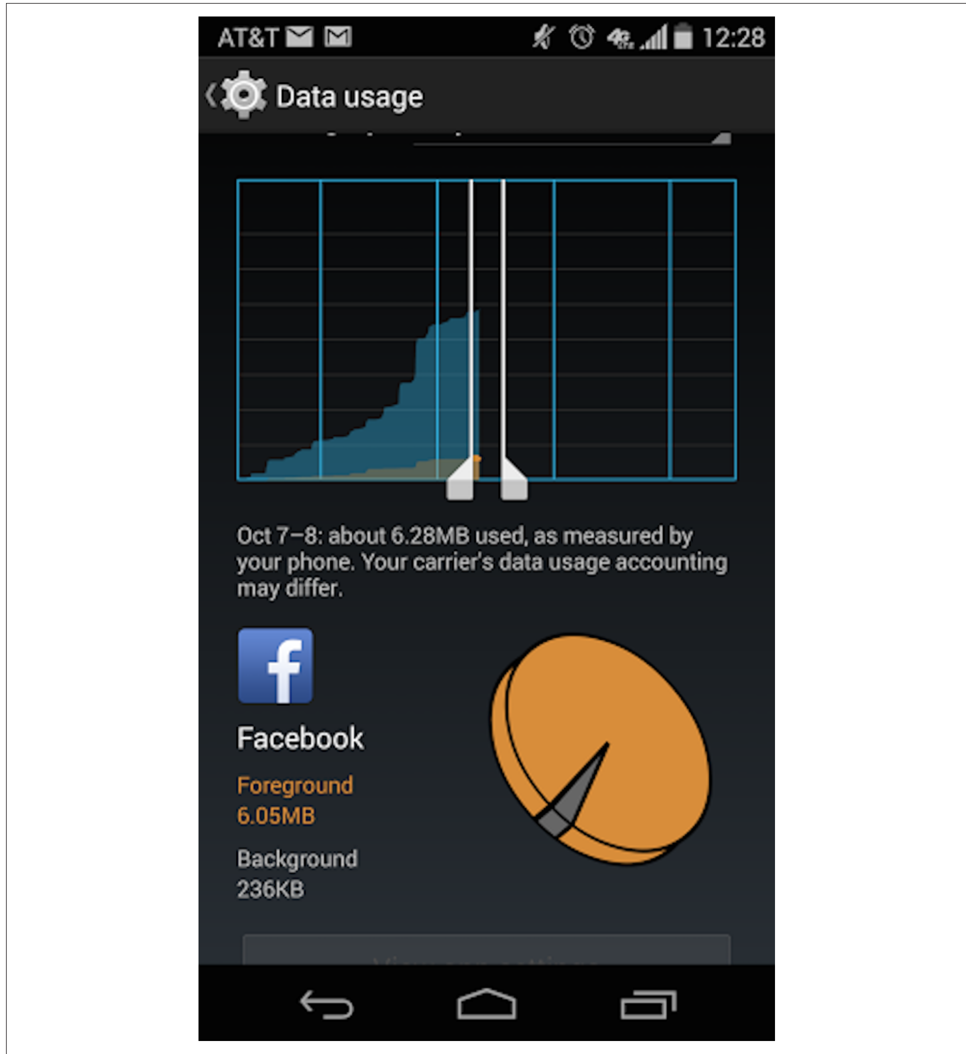


Figure 3-5. Facebook data details, shown in KitKat

In Lollipop, this menu is again changed, with the loss of the sliders that allow you to change the measurement dates. In order to do the analysis I am about to show in Figure 3-6, you'll need to remember to reset the data usage graphs before each test in order to only show the data used during the time the test was run.

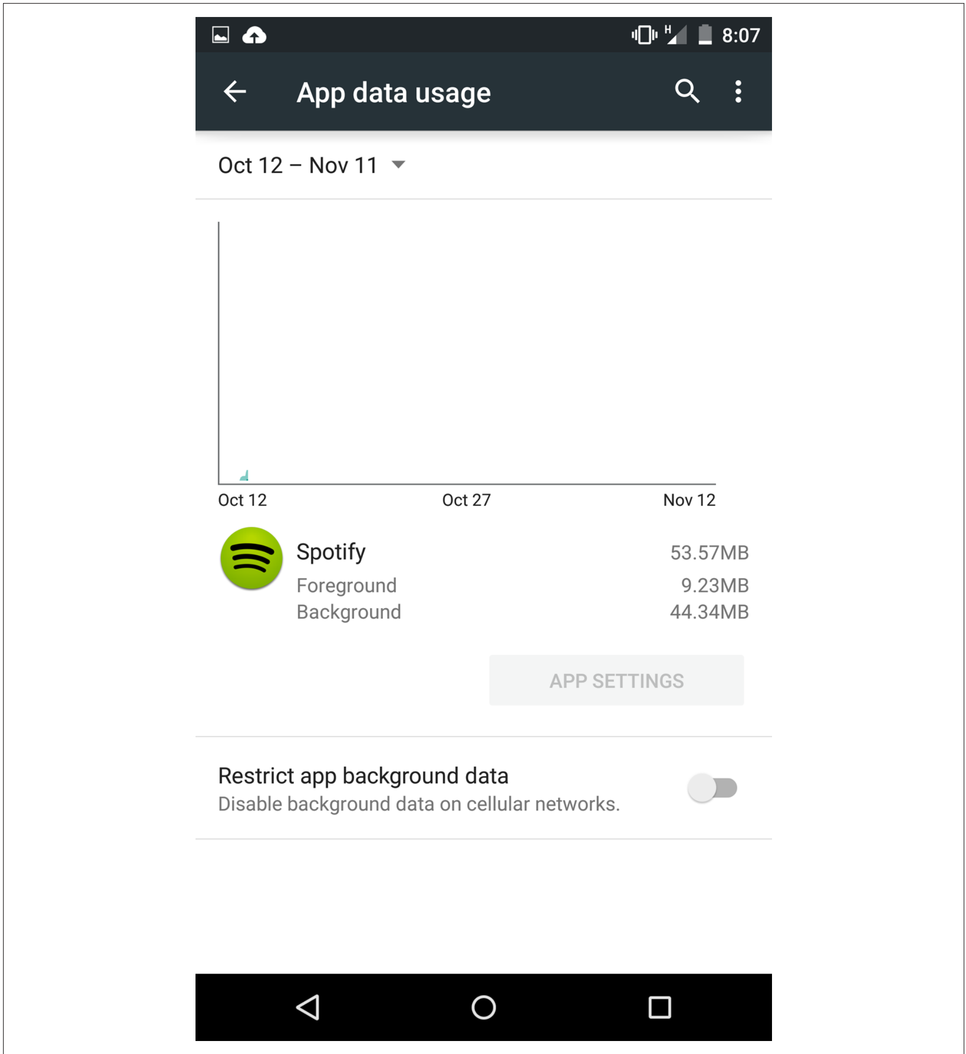


Figure 3-6. Spotify data details, shown in Lollipop

Prior to generating Figure 3-6, I had to reset the phone cellular data totals, so that I could compare the KB of traffic with the number of packets. Comparing the data usage here to the packet data in Figure 3-4, we now know that the 40,024 packets received (from the battery menu) delivered 53.57 MB of data, giving us useful values like 1,403 bytes/packet, and 23.6 kb/s. This is a pretty dense data traffic pattern (which is expected for an app streaming music). If you find your phone has apps that are using lots of packets with low byte count or low throughput rates, you may want to consider disabling data (or perhaps disabling background data) for these apps. These apps may be using the radio inefficiently, potentially causing extra power drain.

Careful monitoring of the battery can help you find mobile apps that are using more data than you'd expect, and based on the data you collect, you can decide to keep or delete the app. By combining the information from the various menus, you can discern a great deal about the battery drain and the data consumption of your mobile app. Today, this takes a real concerted effort by the end user to determine battery hogging apps, but it is only a matter of time until these comparison tools become easier and more mainstream.

App Standby

In Marshmallow, Google has announced a new feature called App Standby. App Standby will prevent infrequently used apps (i.e., ones that have not been used for several days) from connecting to the network or running any processes until the device is plugged in. As a user, this means that these rarely used apps will not drain the battery, lengthening your daily time between charges.

To see a list of app usage in the last day/week/month/year, `adb shell dumpsys usagestats` will tell you the processes, and when they were last active. To see a list of apps and if they are currently active or inactive (have been placed on App Standby), there is a new "inactive apps" setting in Developer Options.

Advanced Battery Monitoring

The initial tests I have described to monitor app battery usage just use various Android Settings menus. They are great for a high-level measurement of battery drain, and can be useful to discover poorly behaving apps on your device. However, from a developer's perspective, they do leave a lot to be desired. In KitKat, Android added the `batterystats` system dump (this is the reason wakelock reporting stopped working in all the Google Play wakelock monitoring apps). This has been expanded in Lollipop to provide more information and some visualization tools have been added as well.

batterystats

`batterystats` is a huge data dump of information about how the device (and all of the running processes) utilize the battery. Introduced in KitKat, it was updated with a large dataset in Lollipop (including data on every wakelock action taken on the device). Before collecting the trace, it is always good to reset the data, and to obtain the most data you can, turn on full wakelock reporting (Lollipop and newer only):

```
adb shell dumpsys batterystats --reset

adb shell dumpsys batterystats --enable full-wake-history
```

Note that resetting `batterystats` also resets all of the data in the Battery Settings menu.

To get an idea of what a `batterystats` system dump looks like, initiate a `batterystats` dump into your command-line interface (in this case, downloading all the data since the phone was last fully charged):

```
adb shell dumpsys batterystats --charged
```

As the reams of data scroll past your terminal, you can tell that there is a lot of information here, but what does it all mean? Let's walk through a few useful sections from the output stream. We'll be able to see some basic stats of the device—how long the device was in different radio states, how much data was sent, and how long the device was kept in full or partial wakelocks.

The following excerpts are from a 30-minute trace `batterystats` dump where I played a game (and as we'll see, got a Facebook message), and then let the phone idle. The first table shows that the battery lost 1% of battery every 2 minutes (or so). The chart is read from bottom to top (my phone started at 97% and ended at 88%). The drain is pretty constant, but you could imagine seeing different timings if the device was idle rather than in use:

```
Discharge step durations:
  #0: +2m28s313ms to 88 (screen-on, power-save-off)
  #1: +2m38s364ms to 89 (screen-on, power-save-off)
  #2: +2m27s323ms to 90 (screen-on, power-save-off)
  #3: +2m8s449ms to 91 (screen-on, power-save-off)
  #4: +2m17s115ms to 92 (screen-on, power-save-off)
  #5: +2m7s924ms to 93 (screen-on, power-save-off)
  #6: +2m17s693ms to 94 (screen-on, power-save-off)
  #7: +2m6s425ms to 95 (screen-on, power-save-off)
  #8: +1m50s298ms to 96 (screen-on, power-save-off)
  #9: +3m0s436ms to 97 (screen-on, power-save-off)
```

The next table contains device statistics. We can see that the phone was on battery for just over 30 minutes, the screen was off for 3.5 of those minutes, but the device was awake for 52 seconds while the screen was still off. The screen was on for 27 minutes, and the brightness was set to dark (the background was dark, and brightness set at ~40%). The cellular signal bounced around from none to good, but mostly in the poor to moderate range. There was Level 4 power WiFi available, but the WiFi was off:

```
Statistics since last charge:
  System starts: 0, currently on battery: false
  Time on battery: 30m 36s 621ms (99.3%) realtime, 27m 58s 456ms (90.8%) uptime
  Time on battery screen off: 3m 31s 100ms (11.4%) realtime, 52s 935ms (2.9%) up
```

```
Total run time: 30m 48s 839ms realtime, 28m 10s 674ms uptime
Start clock time: 2014-10-17-22-54-33
Screen on: 27m 5s 521ms (88.5%) 1x, Interactive: 27m 5s 837ms (88.5%)
Screen brightnesses:
  dark 27m 5s 521ms (100.0%)
Total full wakelock time: 29m 16s 938ms
Total partial wakelock time: 17s 153ms
Mobile total received:187.99KB, sent:201.15KB (packets received 750, sent 742)
Phone signal levels:
  none 35s 29ms (1.9%) 10x
  poor 11m 7s 494ms (36.3%) 96x
  moderate 18m 29s 647ms (60.4%) 94x
  good 24s 451ms (1.3%) 7x
Signal scanning time: 0ms
Radio types:
  hspa 15m 12s 768ms (49.7%) 49x
  hspap 15m 23s 853ms (50.3%) 49x
Mobile radio active time: 14m 32s 106ms (47.5%) 41x
Mobile radio active unknown time: 1m 23s 222ms (4.5%) 21x
Mobile radio active adjusted time: 0ms (0.0%)
Wi-Fi total received: 0B, sent: 0B (packets received 0, sent 0)
Wifi on: 0ms (0.0%), Wifi running: 0ms (0.0%)
Wifi states: (no activity)
Wifi supplicant states:
  disconn 30m 36s 621ms (100.0%) 0x
Wifi signal levels:
  level(4) 30m 36s 621ms (100.0%) 0x
Bluetooth on: 0ms (0.0%)
Bluetooth states: (no activity)
```

The following section titled "Device battery use since last full charge" shows the estimates of battery usage % over the time period. This report is fairly boring, because the battery drain was steady on constant through the trace. I have seen this table display results with several percentage point differences. The last row, telling you how much power is drained when the screen is off, can be a red flag for apps running in the background:

```
Device battery use since last full charge
  Amount discharged (lower bound): 10
  Amount discharged (upper bound): 11
  Amount discharged while screen on: 11
  Amount discharged while screen off: 0
```

The last table shown is only partially replicated here due to length. It shows all of the processes that drew power, and the breakdown from the total power drain:

```
    Estimated power use (mAh):
      Capacity: 3220, Computed drain: 359, actual drain: 322-354
      Uid u0a117: 106
      Screen: 96.6
      Uid 1000: 26.1
      Uid 0: 24.9
      Cell standby: 22.9
  ...
```

As we move past device-specific data, we begin to get more app-specific data, starting with a list of each process's radio usage. For each process, we can see ms per packet (mspp—how frequently the packets arrive—which is a representation of efficiency). For efficient data consumption, mspp should be as low as possible. We are also provided with the packet count and time of radio usage, and the number of times the process turned on the radio. Elsewhere in the report, there are tables that decode the Uid to a human-readable app name (we'll leave these processes anonymous here).

```
    Per-app mobile ms per packet:
      Uid u0a111: 1569 (116 packets over 3m 1s 969ms) 26x
      Uid u0a77: 851 (119 packets over 1m 41s 309ms) 6x
      Uid u0a117: 592 (30 packets over 17s 772ms) 2x
      Uid u0a96: 541 (178 packets over 1m 36s 266ms) 9x
      Uid u0a116: 531 (106 packets over 56s 234ms) 5x
      Uid u0a102: 420 (248 packets over 1m 44s 152ms) 8x
      Uid u0a73: 361 (33 packets over 11s 906ms) 2x
      Uid 0: 339 (113 packets over 38s 347ms) 14x
      Uid u0a10: 335 (389 packets over 2m 10s 380ms) 14x
      Uid u0a28: 239 (160 packets over 38s 221ms) 5x
      TOTAL TIME: 12m 56s 556ms (0.0%)
```

Finally, for each process, the `batterystats` dump lists of all of the wakelocks, and then a breakdown of all data, wakelocks, and power usage for every app, broken down by process. I am only displaying one app (u0a116, which in this case is Facebook Messenger):

```
    u0a116:
      Mobile network: 6.49KB received, 5.94KB sent (packets 63 received, 43 sent)
      Mobile radio active: 56s 234ms (6.4%) 5x @ 531 mspp
      Wake lock *vibrator* realtime
      Wake lock AudioMix realtime
      Wake lock *alarm*: 26ms partial (3 times) realtime
      Wake lock wake:com.facebook.orca/com.facebook.push.mqtt.receiver.MqttReceiver
      TOTAL wake: 26ms partial realtime
      Vibrator: 100ms realtime (1 times)
      Foreground for: 1m 10s 792ms
      Active for: 30m 36s 621ms
      Proc com.facebook.orca:
        CPU: 1s 160ms usr + 470ms krn ; 0ms fg
      Apk com.facebook.orca:
        6 wakeup alarms
        Service com.facebook.push.mqtt.receiver.MqttReceiver:
```

```
  Created for: 148ms uptime
  Starts: 7, launches: 7
Service com.facebook.conditionalworker.ConditionalWorkerService:
  Created for: 61ms uptime
  Starts: 1, launches: 1
Service com.facebook.analytics.service.AnalyticsService:
  Created for: 1m 16s 407ms uptime
  Starts: 2, launches: 2
Service com.facebook.orca.chatheads.service.ChatHeadService:
  Created for: 1m 11s 176ms uptime
  Starts: 1, launches: 1
Service com.facebook.push.fbpushdata.FbPushDataHandlerService:
  Created for: 52ms uptime
  Starts: 2, launches: 2
Service com.facebook.orca.notify.MessagesNotificationService:
  Created for: 540ms uptime
  Starts: 4, launches: 4
```

While I was recording this trace, I got a Facebook message from my spouse. This table gives us a wealth of information as to what Facebook Messenger did for this one simple process of receiving a message:

- The cellular radio was on for ~56 seconds to receive 6.49 KB and to send 6 KB.
- The phone used a wakelock to vibrate an alert to me.
- The phone used the audio wakelock to beep at me.
- These alarm partial wakelocks took 26 ms.
- The vibration was 100 ms of shaking, but is independent of the wakelock.

Facebook Messenger runs in the background all of the time. Battery Historian shows that while active for the full 30 minute 36 second trace, it was only in the foreground for 1 minute 10 seconds, and the CPU time was only 160 + 470 = 630 ms. Basically, the app was sitting in wait for a message to arrive, and when a message arrived, it woke up for a minute to alert me and do work.

The one minute of usage was primarily used by the `ChatHeadService` and `AnalyticsService`. The `ChatHead` opens a bubble in the foreground of my device, indicating that I received a message. It was active for 1 minute in case I wanted to message back. The `AnalyticService` was open for a few extra seconds after the `ChatHeadService` ended in order to report that indeed, I did not respond back.

Battery Historian

The details in the `batterystats` output are extremely helpful in determining how mobile apps behave with the different battery-consuming aspects of the device. When drilling into details for one app, there is an extensive amount of detail that is very useful to understand how the app is behaving and to uncover potential issues. However, digging through long text outputs is time consuming and feels somewhat like looking for a needle in a haystack. To simplify the analysis, Google has created Battery Historian, a script that takes the raw `batterystats` output file, and charts the data into an HTML document. At its simplest, you can run the following commands to create a web page that visualizes the information from `batterystats`:

```
adb bugreport > bugreport.txt  //download the output to your computer
./historian.py  bugreport.txt > out.html //create the html file
```

However, at Google I/O 2015, Battery Historian 2.0 was launched. This new report was completely rewritten in GO, and provides more information to help you drill into the battery data for your specific app (note that your device must be on Lollipop or newer OS). To begin, let's look at the report that is identical in both versions of Battery Historian (labeled Historian-Legacy in version 2.0).

We'll continue evaluating the same trace, but now in the browser. As shown in Figure 3-7, the Battery Historian chart includes a *lot* of data recorded by your device.

www.allitebooks.com

Figure 3-7. Battery Historian-Legacy (main view)

As we have shown, the raw data file was complicated and had a lot of data. This table, while simplified, is still complicated and deserves a walkthrough. Zooming into the top of the report, we see the information presented in Figure 3-8.

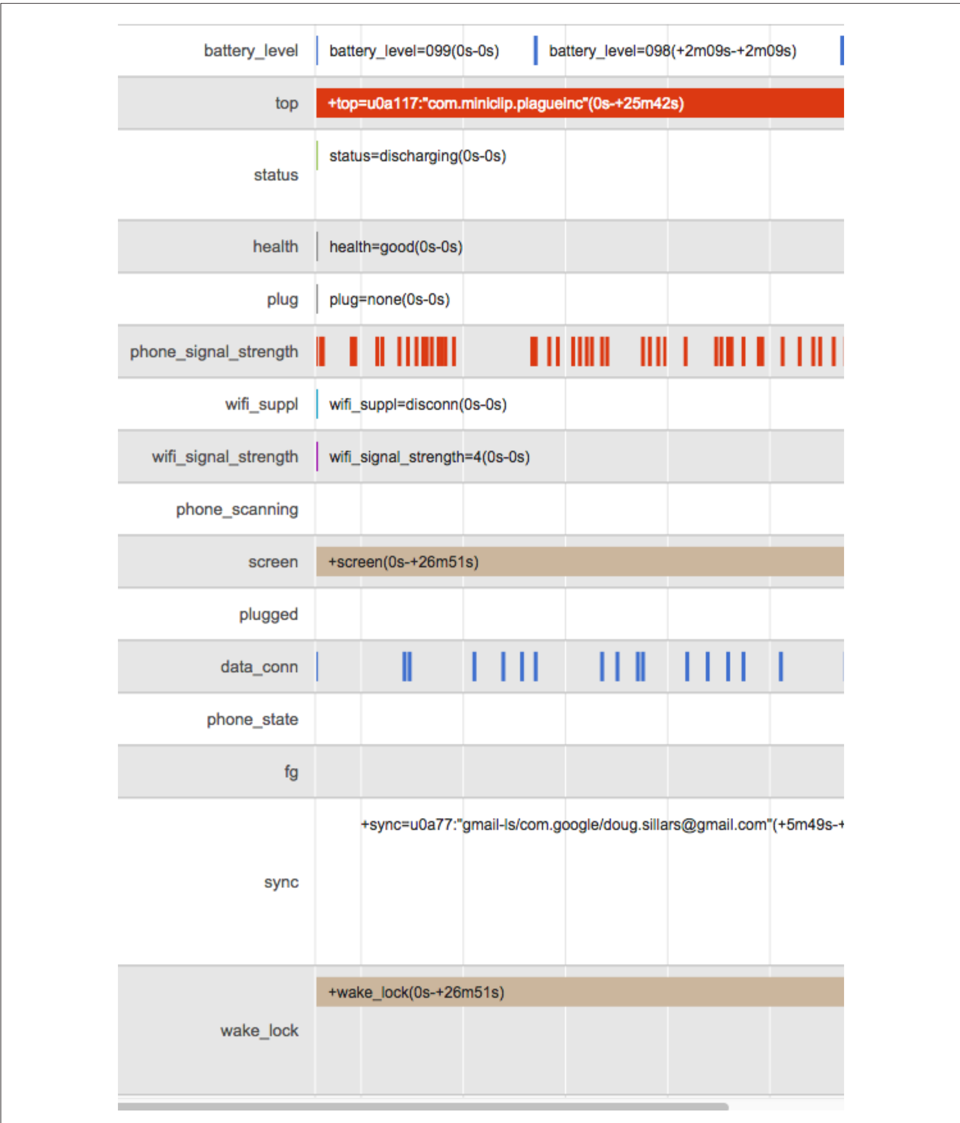


Figure 3-8. Battery Historian-Legacy (top view)

In Figure 3-8, the white vertical bars indicate 1-minute intervals. For any item in the chart, mousing over provides more details:

battery_level
: Mousing over a battery level change provides the level of the battery, and how long it has been since the last change of battery level (as seen in Figure 3-9).

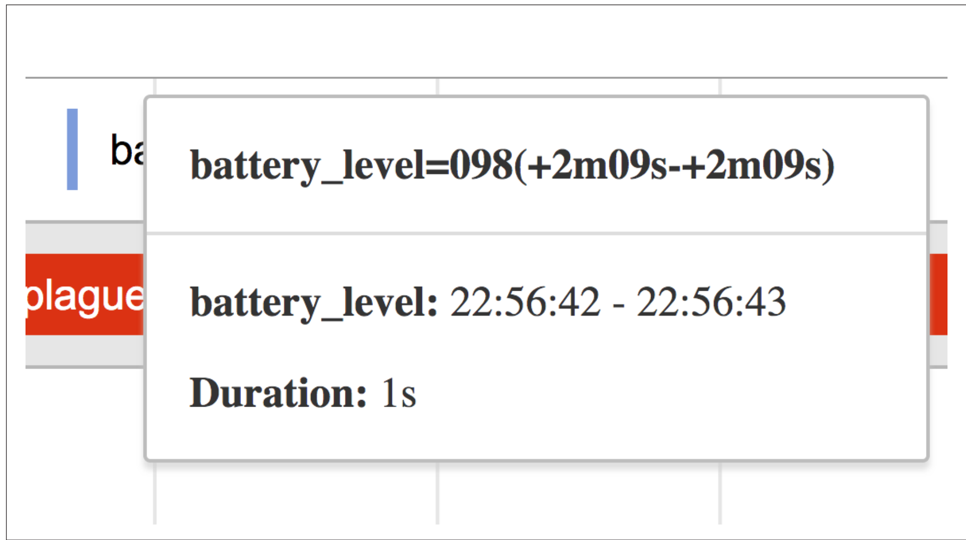


Figure 3-9. Battery level change

`top`
: Lists the process that was actually on the screen (in this example, the game Plague Inc.)

`Battery info`

- `status`
  : Battery is discharging (as opposed to being plugged in).
- `health`
  : Battery health from the Battery Manager API.
- `plug`
  : Is device plugged in?

`phone_signal_strength` *(radio information)*
: Shows changes in cellular signal (in this case, from poor, moderate, and good).

Mousing over a change in cellular signal strength tells you information about the strength of the signal (as seen in Figure 3-10).

Figure 3-10. Cellular signal strength

`wifi_suppl`
: In Figure 3-8, it is disconnected.

`wifi_signal_strength`
: In Figure 3-8, the Wi-Fi signal is detected—even with Wi-Fi off, due to the Advanced Wi-Fi setting to always scan.

`phone_scanning`
: If there is no signal, the phone will scan (using more battery power).

`screen`
: On/off and duration on.

`plugged`
: Power source (similar to the Battery data above).

`data_conn`
: Mousing over the blue connections shows the cellular data switching from HSPA to HSPAP.

Mousing over data_conn provides details about the type of data connection the cellular radio has in place (as seen in Figure 3-11).

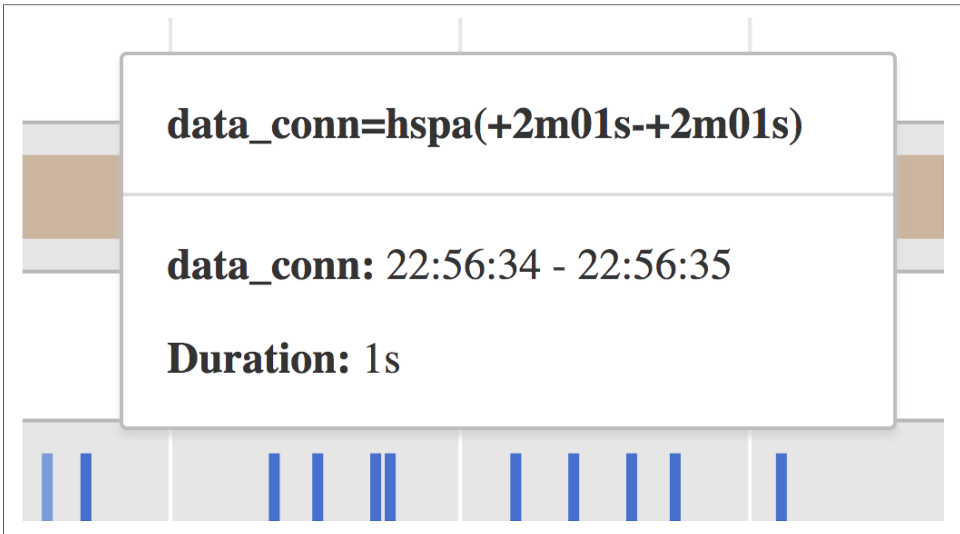


Figure 3-11. Data connection type

`phone_state`
: Shows changes in cellular coverage, or if you get a phone call.

`fg`
: This refers to foreground apps. Apps running in the foreground are less likely to be killed to relieve memory pressures. Just off the screen, we can see that Facebook was coming into the foreground to process the message I received.

`sync`
: Processes syncing with the server.

Mousing over a sync event provides information about what caused the sync to occur (as seen in Figure 3-12).

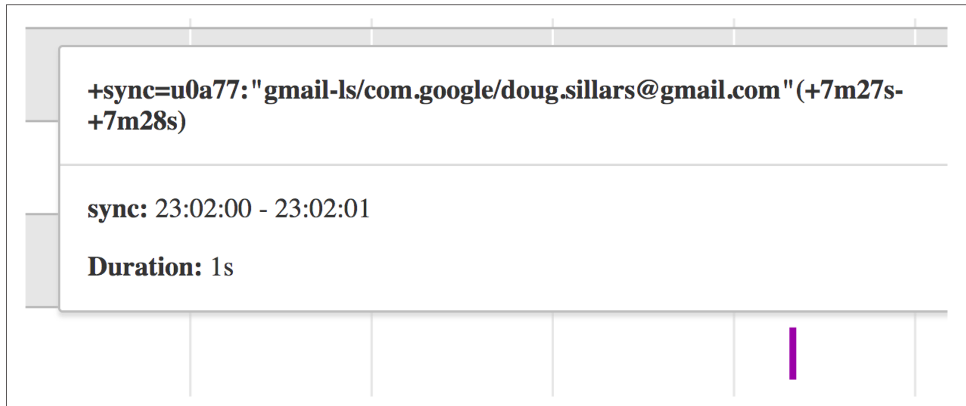


Figure 3-12. Sync causes

Most of these are pretty self-explanatory processes on the device. They set up the analysis of battery drain by giving you the knowledge of the state of the phone (and what apps/calls are being made).

One thing to look at in this view for battery life is how often syncs and wakelocks are occurring. If your mobile app is waking up the device often (and you will likely see your process name in the mouseover information), you should examine the frequency of the syncs and device wakeups. It is important to find the correct balance between up-to-date information and battery life. If you use inexact alarms or the JobScheduler API to wake up the device, you may see multiple syncs or wakelocks happening at once. This is a signal that you are doing things correctly: your alarms are being triggered when other apps wake up the device—thereby minimizing the number of wakeups.

As we progress down the screen, we see more information about the wakelocks and issues that lead to battery drain (Figure 3-13).

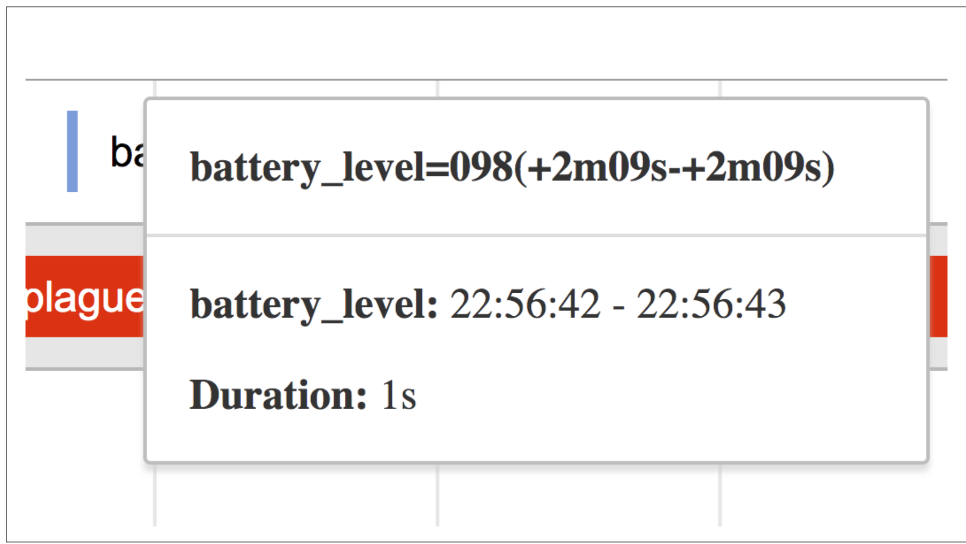


Figure 3-13. Battery Historian-Legacy (bottom view)

`wake_lock`
: There is one prolonged wakelock keeping the screen on during the game.

`gps`
: When the GPS radio turns on.

`running`
: The phone was clearly running, as I was playing a game.

`wake_reason`
: This is when the device wakes from sleep. There are no reasons on this screenshot, because the device was awake for the entire trace. This row lists all of the deep processor-level processes that are running on your device. Some common wake reasons include the following:

  `qcom,smd-modem`
  : Qualcomm Shared Memory Driver interacting with the modem memory.

  `qcom, smd-rpm`
  : Qualcomm Shared Memory Driver - Resource Power Manager.

  `qcom, mpm`
  : Qualcomm MSM Sleep Power Manager; shuts down clock and puts the device to sleep.

`qcom, spmi`
: Qualcomm System Power Management Interface; also working to put the device back to sleep.

`wake_lock_in`
: Here we can see what processes are running and causing the wakelock or alarm to occur. In the screenshots, there are many wakelocks from the audiomix process in the game (nearly 2,000 audio wakelocks occurred as various samples were played). We also see the screen wakelock (fifth row has a line of solid pink).

Mousing over a wakelock event will tell you what process caused the wakelock to occur (as seen in Figure 3-14).

Figure 3-14. Wakelock occurrence

`mobile_radio`
: Time that the cellular radio is connected (not necessarily transmitting, but connected to a network). There are gaps as the phone jumps from different flavors of HSPA.

`user`
: For cases where multiple user accounts might be used.

`userfg`
: Tells you which user is in the foreground during testing.

In Figure 3-13, we are getting to what is waking up the mobile device. As mentioned earlier in "Wakelocks" on page 36, when your app wakes up the device, it leads to battery drain. This screenshot is relatively boring, because a game was underway. We can look at some other traces to identify interesting wakelock phenomena.

Finding bad wakelocks with Battery Historian

If you suspect that your app is causing excess wakelocks, you can use Battery Historian to verify. While running a long Battery Historian trace (the vertical bars indicate 30-minute intervals in Figure 3-15), I force stopped an app that was using too many wakelocks.

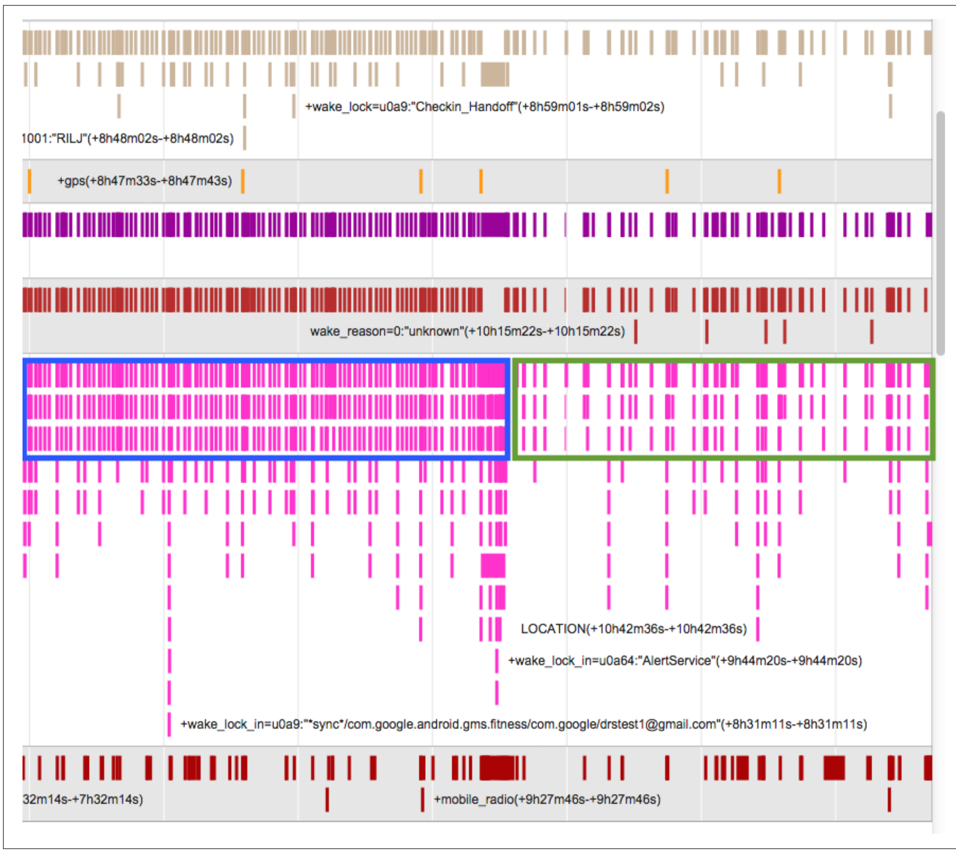


Figure 3-15. Finding excess wakelocks with Battery Historian

Qualitatively, Battery Historian makes it very easy to spot a change in the wakelock behavior of the device. It is very clear that prior to killing the app in question, there is a frequent repeat pattern to a number of wakelocks, as seen in the blue box. The app was firing at least three wakelocks per minute: the app, location, and accelerometer. These wakelocks are always 1 minute apart. In between the blue and green box, I stopped the app, and immediately, the quantity and frequency drop (growing to as much as 5 minutes between wakelocks), as seen in the green box.

Below the Battery Historian chart is a list of all of the events seen during the trace. In the case of this rogue app, by running a trace with the app running—and one without

—I can calculate a drop from 594 events/hour to 478 events/hour after stopping process. This implies that ~120 wakelocks per hour were caused by this one app. I don't expect that your apps have this many wakelocks, but it is a good study to ensure that you are not waking up the device too often. As the wakelock and alarm APIs state, it is crucial to be mindful of wakelock behavior, as it has a large effect on battery utilization of Android devices.

Battery Historian 2.0

With the release of Battery Historian 2.0 (BH2), the Android team completely rewrote the tool (from Python into Go). The new version has all of the views shown in the previous section, but adds even more functionality that allow you to go deeper than the device level and interrogate the battery usage of each process. Rather than creating a web page through a script, you create a service running on port 9999 that will parse and display the report for you. Let's take a quick look at the new features in BH2. When you open a bugreport file, there is a new UI, as shown in Figure 3-16.

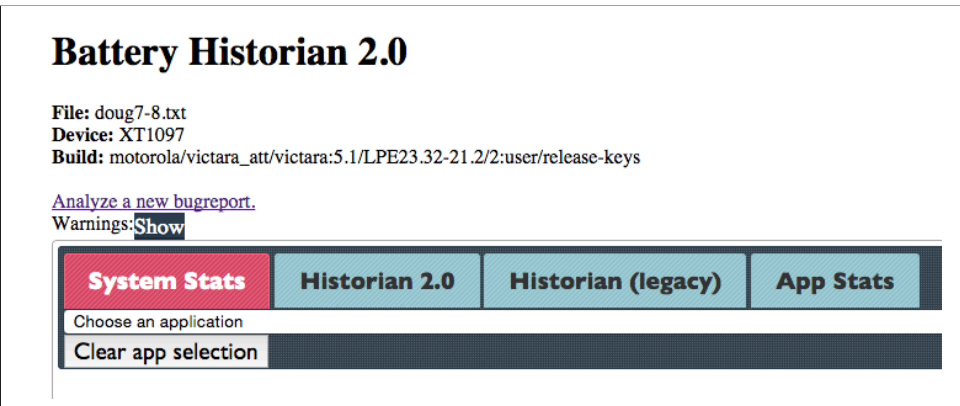


Figure 3-16. Battery Historian 2.0 header

The header shows the name of the file, the device (Moto X 2014 running Lollipop 5.1), and has four tabs: System Stats, Historian 2.0, Historian (legacy), and App Stats. We've already covered the legacy Battery Historian, so let's look at the three new tabs of information.

The System Stats tab consists of a half dozen tables detailing how the battery was drained during the study. The first table lists the aggregate stats (Figure 3-17): in nearly 4 hours, the screen was on for 40 minutes, the device was awake with the screen off for 24 minutes. The battery drain with the screen on was 19%/hour, and almost 4%/hour with the screen off. The cellular radio was on for over 2 hours, and averaged 2.6 MB/hour.

XT1097 LPE23.32-21.2

Aggregated Stats:

Metric	Value
Device	XT1097
Build	LPE23.32-21.2
Duration / Realtime	3h49m32.521s
Screen Off Discharge Rate (%/hr)	3.81 (Discharged: 12%)
Screen On Discharge Rate (%/hr)	19.31 (Discharged: 13%)
Screen On Time	40m23.648s
Screen Off Uptime	24m48.785s
Userspace Wakelock Time	8m16.784s
Kernel Overhead Time	16m32.001s
Mobile KBs/hr	2603.93
WiFi KBs/hr	0.00
Mobile Active Time	2h17m55.355s
Signal Scanning Time	0

Figure 3-17. Battery Historian 2.0 aggregate stats

Looking closely, there are five underlined Metrics in Figure 3-17. Each of these Metrics has a corresponding table where these stats are further broken down by process. For example, let's look at mobile radio uptime and data usage (Figure 3-18).

Mobile radio (active) time per app:

Ranking	Name	Uid	Mobile Active Time
0	com.levelup.touiteur		
1	GOOGLE_SERVICES		
2	com.att.connect		
3	com.google.android.googlequicksearchbox		
4	ROOT		
5	com.facebook.katana		

Mobile traffic per app:

Ranking	Name	Uid	MB Transferred Over Mobile
0	com.att.connect		
1	com.google.android.googlequicksearchbox		
2	com.android.chrome		
3	com.levelup.touiteur		
4	GOOGLE_SERVICES		
5	com.google.android.apps.inbox		

Figure 3-18. Battery Historian 2.0 radio usage statistics

Looking at the process details in Figure 3-18 for mobile radio time/app and kb/app, we can see what apps are using the cellular radio the most during the study. In this case, we see that *com.levelup.touiteur* (a Twitter client) uses the most radio time, but *com.att.connect* uses the most data (for the Moto X, the time/app and KB/app are not populated, but this does populate for other devices; the ranking of apps is still accurate).

That the process *com.att.connect* used a lot of data is not surprising. During this study, I used this app to stream a 30-minute teleconference from a colleague's desktop. I was surprised, however, to see that my Twitter app was online for longer than my teleconference. Compiling the data from the App stats tab, see in the following table that the Twitter app connected more times, sent less data, but used more radio and battery time than my teleconference app.

App	# connections	KB	Mobile time	%Battery
com.levelup.touiteur	18	1014	23 min	5.91%
com.att.connect	6	3056	18 min	5.78%

The app stats page also lists the wakelocks (and their duration), services, and processes each app used during the trace. My Twitter client had 18 partial wakelocks (at least they all overlapped), had 35 services wakeup, and used 2 processes. These are very powerful ways to dig into the way your app behaves, but also a very nice report format to share with your teams.

Stepping back to the entire trace for a moment, the Historian 2.0 tab has a new UI showing how the wakelocks and device usage affect battery life (Figure 3-19).

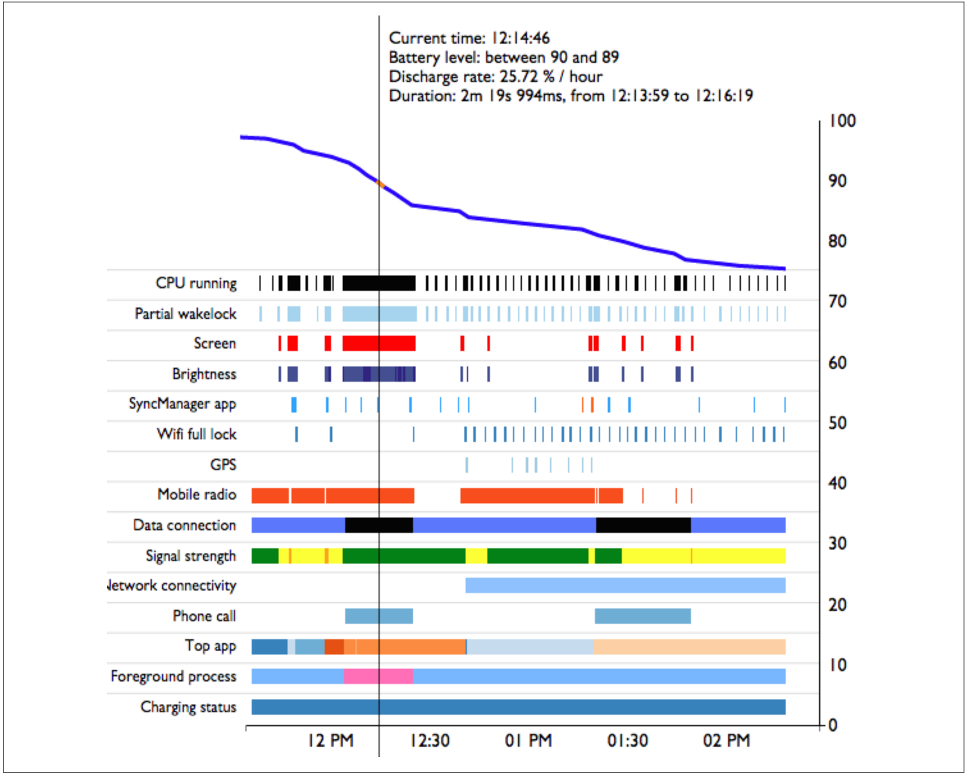


Figure 3-19. Battery Historian 2.0 graph

This new chart has familiar bars for wakelocks, CPU, GPS, radio, and so on, but also adds a new axis on the right side, and a blue bar overlaid on top of the data visually indicating the battery drain. Each 1% segment of the battery drain is selectable, and stats on the drain over this period are presented. In the close-up shown in Figure 3-20, the vertical line indicates the battery drain section. This very rapid drain (1% of battery in 2 minutes or about 25%/hour) was when I was streaming the teleconference, and listening on the phone.

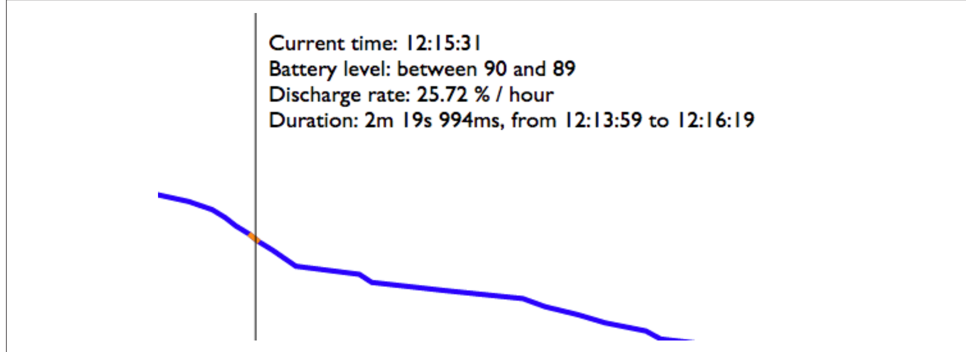


Figure 3-20. Battery Historian 2.0 battery drain detail

The original Battery Historian tool gave a lot of device-level stats that helped determine how individual apps behave. The additions in Battery Historian 2.0 make digging into data for one single process a much simpler task. Now you can very easily isolate the battery drain functions of your app and from that work to resolve the issue.

JobScheduler

In Lollipop, Android added a new API called JobScheduler. It is a new framework that can be used instead of wakelocks and alarms to run jobs for your app. Think of it as "a wakelock/alarms that plays well with others" API. While wakelocks and alarms are app specific, the JobScheduler abstracts the device wakeups to the OS. Because alarms and wakelocks are sandboxed to your app, there is no way to coordinate these with the other apps installed on the device. If five apps wake up every 30 minutes, their alarms are unlikely to by synced, resulting in 10 wakeups per hour. Because JobScheduler abstracts the wakeup to the system, the system can piggy-back all of the scheduled jobs in an efficient way, so that there might only be a few wakeups per hour.

In addition to scheduling future wakeups, the JobScheduler allows you to supply a time range where after 8 minutes it is OK to get the data, but it *must* be collected by 10 minutes. Providing a range allows the OS to better coordinate to save battery. It also means that your app may get required data earlier than required, but in a way that saves battery (which is a win-win for your app)! Imagine your weather app that connects every 10 minutes (6x per hour). However, if the radio is on, does it really matter if one update happens early if it saves battery? In many cases, this just means that that data is updated faster than the requirement of the app, but uses less battery as a result. In the following code snippet (derived from my modified JobScheduler app), I set the minimum time between connections at 7 minutes, but force a connection at 10 minutes (when the deadline is reached):

```
JobInfo.Builder builder = new JobInfo.Builder(kJobId++, mServiceComponent);
      //kJobId allows me to run multiple JobScheduler runs at the same time
      <snip>
            String delay = mDelayEditText.getText().toString();
              //read delay time(s) from UI
              if (delay != null && !TextUtils.isEmpty(delay)) {
                  builder.setMinimumLatency(Long.valueOf(delay) * 1000);
              }
              String deadline = mDeadlineEditText.getText().toString();
                              //Read deadline time from UI
              if (deadline != null && !TextUtils.isEmpty(deadline)) {
                  builder.setOverrideDeadline(Long.valueOf(deadline) * 1000);
           }

   boolean requiresUnmetered = mWiFiConnectivityRadioButton.isChecked();
   boolean requiresAnyConnectivity = mAnyConnectivityRadioButton.isChecked();
   if (requiresUnmetered) {
       builder.setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED);
   } else if (requiresAnyConnectivity) {
       builder.setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY);
   }

   builder.setRequiresDeviceIdle(mRequiresIdleCheckbox.isChecked());
                      //checkbox to force JS only when idle
   builder.setRequiresCharging(mRequiresChargingCheckBox.isChecked());
                      //checkbox to force JS only when charging
   mTestService.scheduleJob(builder.build());
```

Other helpful features of the JobScheduler API can be seen in the code (they are powered by checkboxes):

- Run a periodic service, with a guarantee of a connection *sometime* in the periodic window
- Only run the job on unmetered networks (generally Wi-Fi)
- Only run when the device is idle (the API is pretty non-specific on what idle means, only saying the device has not been in use for "some time")
- Run when the device is plugged in
- Fallback of connections; increase the time between subsequent connections

Imagine that your app wakes up your customers' device every 15 minutes to check for updates on the server. That would be four wakeups an hour (96/day). What if your customers also have a weather app that updates every six minutes (10x/hour, 240x/day)? The odds of your app and the weather app connecting at the same time are extremely low—because your timers are not synchronized, and there is no interaction between the apps. If both apps had used the JobScheduler API, the OS would coordinate to save power. In their Project Volta presentation at Google I/O 2014, Google estimated 15%–20% battery savings if every app used this API.

I have extended the Android SDK JobScheduler sample app to allow interaction with some of its additional features. The job is to download an image from a server. In Figure 3-21, I have set the app to download an image every 60 seconds (the API will make a connection inside a timing of 60 s, but not necessarily at exactly 60 s). This means that in ~14 minutes the app will ping 14 times.

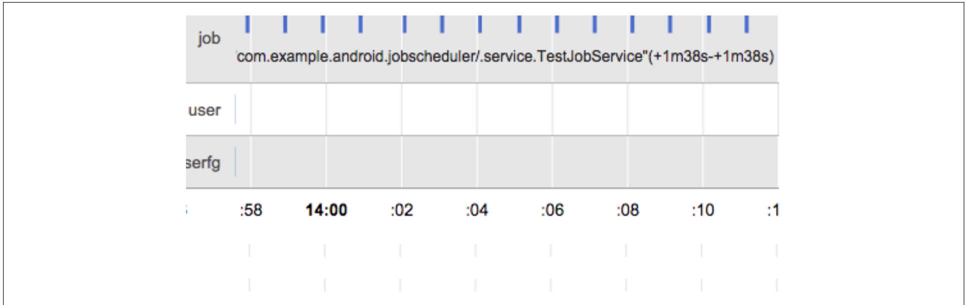


Figure 3-21. Periodic connection in JobScheduler (set to 60 seconds)

Figure 3-22 shows a similar test, but where the periodicity is set to download every 150 s (same scale), resulting in six connections over 14 minutes.

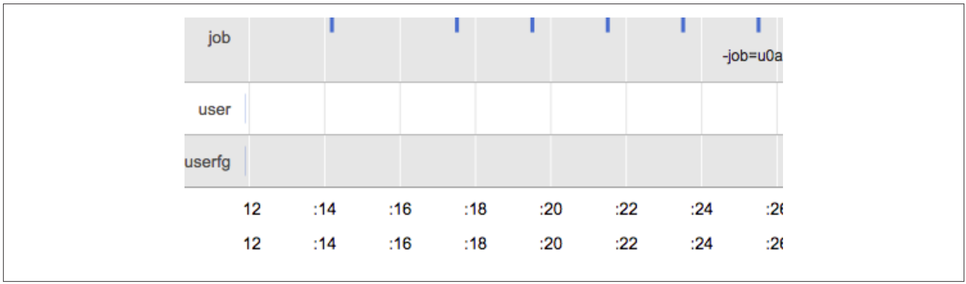


Figure 3-22. Periodic connection in JobScheduler (set to 150 seconds)

With traditional wakelocks, we'd assume that if these were sandboxed apps running simultaneously, there would be 20 connections made by the device, as it is unlikely that the connections would overlap. However, when we run these two jobs simultaneously with JobScheduler, the system has synced up the periodic connections to reduce the amount of battery drain. Instead of 20 connections, the same data is transferred in just nine connections.

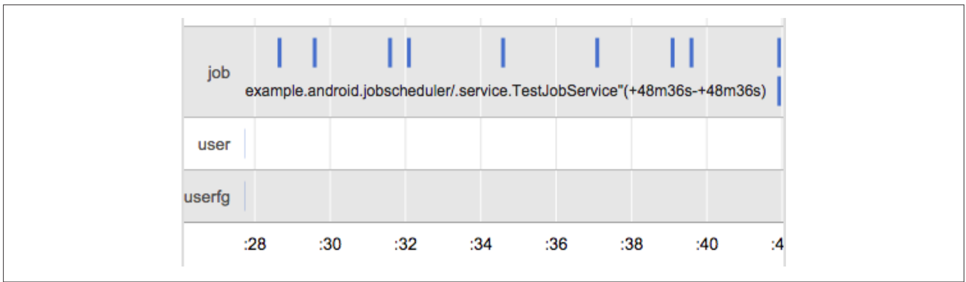


Figure 3-23. Synced 60 s and 150 s connections

Another cool feature of JobScheduler is the ability to repeat the job, but with a linear or exponential backoff. If your app is not in the foreground, the need to continue getting frequent updates is diminished, so you can allow them to become less frequent. When the app is reopened, that data will still be fresh for your customers, but with less background data usage. The JobScheduler fallback has two options for delays between jobs: linear (for slowly backing off) or exponential (faster backing off). The linear fallback takes the current deadline, and adds fallbacktime*(number of failures -1). In the example shown in Figure 3-24, the deadline was 20 s , and the fallback delay was another 20, so each subsequent ping adds 20 s (sched2start differences of 20, 40, 60, etc.). The exponential backoff adds fallbacktime*2^(number of failures -1), so the delay time grows by a power of 2 between each ping (sched2start difference of (20, 60, 120, 180, 325, 645). Now your app data is being updated for your customer, but in a way that uses the network less—saving battery.

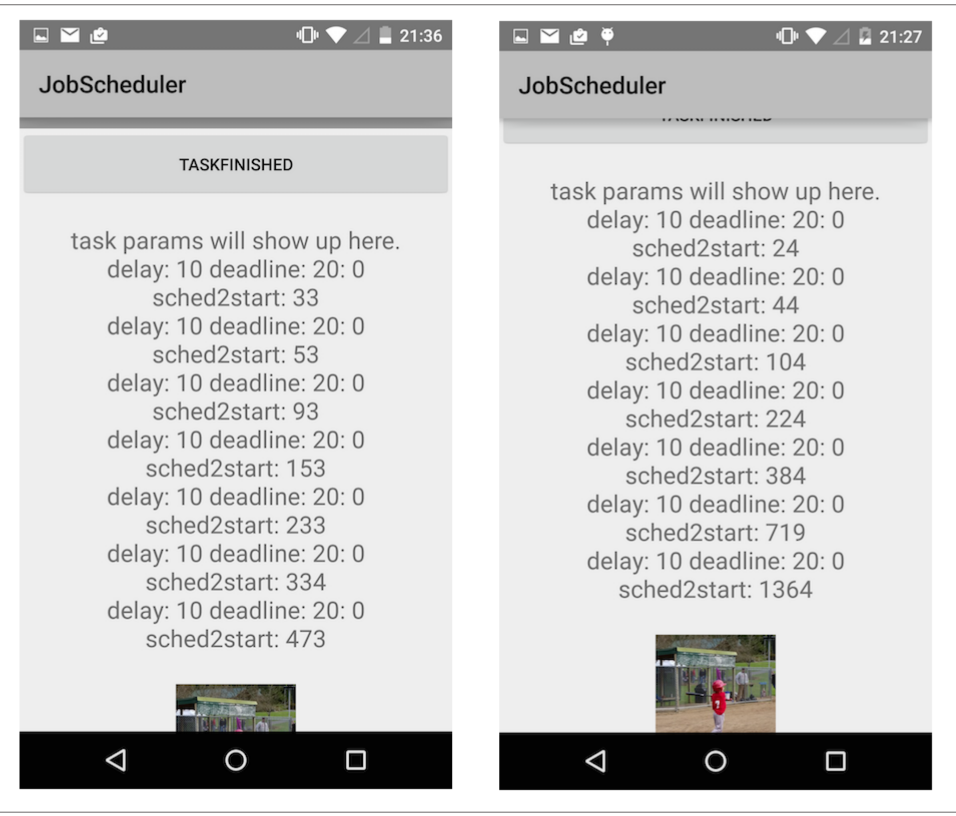


Figure 3-24. Screenshot of the JobScheduler app with linear (left) and exponential (right) fallback

It is clear that allowing the OS to schedule non-critical jobs is a great way to conserve battery life in your app. At the time of this writing (April 2015), Lollipop is just 5.5% of all Android devices, but as this population grows, more of your customers will benefit from your adoption of the JobScheduler API.

Conclusion

Battery life is an excellent indicator of app performance. Apps with poor battery life often exhibit symptoms like waking up the device, or not letting the device go back to sleep. We've walked through how Android calculates the power drain for apps (based on hardware measurements), and how customers can discover apps that are causing issues (and help you prevent your app from appearing in this list to your end users). We also looked at the new Battery Historian tools in Android Lollipop that provide more in-depth developer perspective into battery drain in Android apps, and discovered an app that was waking up the device in an excessive manner. Additionally, we explored how the JobScheduler API will reduce the number of background calls by letting the OS run the scheduling of many apps at once.

CHAPTER 4

Screen and UI Performance

The user interface of your app is likely influenced by designers, developers, usability studies, and testing—just about anyone is happy to add input/feedback to how your app looks. As the UI of your app is your connection to your customers, it defines your brand and it requires careful planning. However simple (or complicated) the UI of your app is, it's important that your UI design is built to be performant.

As a developer, your task is to work with the UI/UX team and build an app that follows its design parameters on every Android device. We've already (briefly) discussed the pitfalls of the many screen sizes in the Android ecosystem and the challenges that exist there. But how about UI performance? How does the UI that your designers designed (and you built) run? Do the pages load quickly? Do they respond in a fast and smooth way? In this chapter, we'll discuss how to optimize your UI for fast rendering and scrolling/animations, and the tools you can use to profile your screen and UI performance.

UI Performance Benchmarks

Like all performance goals, it is important to understand the performance goals associated with UI. Saying "my app needs to load faster" is great, but what are the expectations of the end user, and are there concrete numbers you can apply to those expectations? In general, we can fall back on studies of the psychology of human interactions. These studies have shown that users perceive delays of 0 – 100 ms as instantaneous and delays of 100 – 300 ms as sluggish; delays between 300 – 1,000 ms indicate to users that "the machine is working," whereas delays of 1,000+ ms lead users to feel a context switch.

As this is basic human psychology, it seems to be a good metric to start with for page/view/app loading times. Ilya Grigorik has a great presentation about building mobile

websites to take just "1,000 ms to Glass." If your web page can load in 1 second, you win the human perception battle, and now you must wow your customers with great content. Additional research has shown that >50% of users begin abandoning websites if no content has loaded in 3–4 s. Applying the same argument to apps tells us that the faster you can get your app to start, the better. In this chapter, we'll focus just on the UI loading. There may be tasks that must run in the background, files to be downloaded from the Internet, and so on, we'll cover optimizing these tasks (or ways to keep these tasks from blocking the rendering) in future chapters.

Jank

In addition to getting content on the screen as quickly as possible, it has to render in a smooth way. The Android team refers to jerky, unsmooth motion as *jank*, and this is caused by missing a screen frame refresh. Most Android devices refresh the screen 60 times a second (there are undoubtedly exceptions—earlier Android devices were sometimes in the 50 or less fps range). Because the screen is refreshed every 16 ms (1 s/60 fps = 16 ms per frame), it is crucial to ensure that all of your rendering can occur in less than 16 ms. If a frame is skipped, users experience a jump or skip in the animation, which can be jarring. In order to keep your animations smooth, we'll look at ways to ensure the entire screen renders in 16 ms. In this chapter, we'll diagnose common issues and illustrate how to remove jank from your UI.

UI and Rendering Performance Updates in Android

One of the major complaints of early Android releases was that the UI—especially touch interactions and animations—were laggy. As a result, as Android has matured, the developers have invested a great deal of time and effort to make the user interaction as fast and seamless as possible. Let's walk through a few of the improvements that have been added in various releases of Android to improve the user interaction:

- On devices running Gingerbread or earlier, the screen was drawn completely in software (there was no GPU requirement). However, device screens were getting larger and pixel density was increasing, placing strain on the ability of the software to render the screen in a timely manner.
- Honeycomb added tablets, further increasing screen sizes. To account for this, GPU chips were added, and apps had the option to run the rendering using full GPU hardware acceleration.
- For apps targeting Ice Cream Sandwich and higher, GPU hardware acceleration is on by default; pushing most rendering out of the software and onto dedicated hardware sped up rendering significantly.
- Jelly Bean 4.1 (and 4.2) "Project Butter" made further improvements to avoid jank and jitter, in order to make your apps "buttery smooth." By improving tim-

ing with VSYNC (better scheduling frame creation) and adding additional frame buffering, Jelly Bean devices skip frames less often. When building these improvements, the Android team built a number of great tools to measure screen drawing, the new VSYNC buffering and jank, and released these tools to developers.

We'll review all of these changes, the tools introduced, and what they mean to the average Android developer. As you might imagine, the goals from these updates were as follows:

- Lower the latency of screen draws
- Create fast, consistent frame rates to avoid jank

When the Android team was working on all of the improvements to screen rendering and UI performance, they needed tools to quantify the improvements that they made to the OS. To their credit, they have included these tools in the Android SDK so that developers can test their apps for rendering performance issues. As we walk through the different ways to improve app performance, we'll use these tools with example apps to explain how they work.

With that said, let's get started!

Building Views

I assume that you are familiar with the XML layout builder in Android Studio, and how to build views with the different tools in Android Studio (Eclipse) to look at those views. In Figure 4-1, you can see a simple app with a series of nested views. When building your views, it is important to look at the Component Tree in the upper-right of the screen. The more nested your views become, the more complicated the View Tree becomes, and the longer it will take to render.

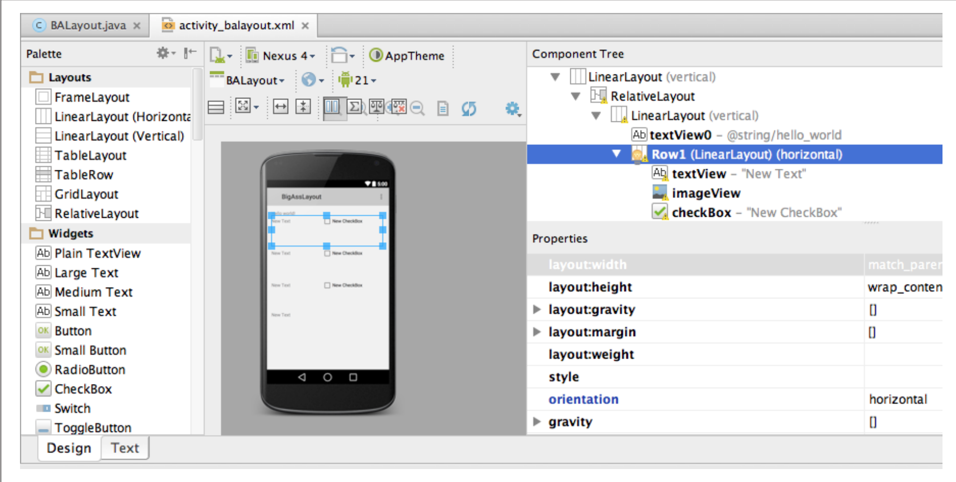


Figure 4-1. Design view of an app layout

For each view in your app, Android goes through three steps to render on the screen: measure, layout, and draw. If you imagine your XML layout hierarchy in your app, the measure starts at the top node and walks the render tree of the layout: measuring the dimensions of each view to be displayed on the screen (in Figure 4-1: LinearLayout; RelativeLayout; LinearLayout; then branching for textView0 and the LinearLayout Row1—which has three further children). Each view will provide dimensions to the parent for positioning. If a parent view discovers an issue in the measurements of its dimensions (or that of its children), it can force every child (grandchild, great-grandchild, etc.) to remeasure in order to resolve the issue (potentially doubling or tripling the measurement time). This is the reason a flat (less nested) view tree is valuable. The deeper the nodes for the tree, the more nested the measurement, and the calculation times are lengthened (especially on remeasurements). We'll examine some examples of how remeasurement can really hurt rendering as we look through the views.

Remeasuring Views

There does not have to be an error for a remeasure to occur. RelativeLayouts often have to measure their children twice to ensure that all child views are laid out properly. LinearLayouts that have children with layout weights also have to measure twice to get the exact dimensions for the children. If there are nested LinearLayouts or RelativeLayouts, the measure time can grow in an exponential fashion (four remeasures with two nested views, eight remeasures with three nested views, etc.). We'll see a dramatic example of remeasurement in Figure 4-9.

Once the views are measured, each view will layout its children, and pass the view up to its parent—all the way back up to the root view. Once the layout is completed, each view will be drawn on the screen. Note that all views are drawn, not just the ones that are seen by your customers. We'll talk about that issue in "Overdrawing the Screen" on page 90. The more views your app has, the more time it will take to measure, layout, and draw. To minimize the time this takes, it is important to keep the render tree as flat as possible, and remove all views that not essential to rendering. Removing layers of the layout tree will go a long way in speeding up the painting of your screen. Ideally the total measure, layout, and draw should be well below the 16 ms threshold—ensuring smooth scrolling of your UI on the screen.

While it is possible to look at the node view of your layout as XML (like in Figure 4-1), it can be difficult to find redundant views. In order to find these redundant views (and views that add delay to screen rendering), the Hierarchy Viewer tool in Android Studio Monitor can greatly help you visualize the views in your Android app to resolve these issues (Monitor is a standalone app that is downloaded as a part of Android Studio).

Hierarchy Viewer

The Hierarchy Viewer is a handy way to visualize the nesting behavior of your various views on a screen. The Hierarchy Viewer is a great tool to investigate the construction of your view XML. It is available in Android Studio Monitor, and requires a device with a developer build of Android on it. See "Rooted Devices/Engineering/Developer Builds" on page 15 for details on what this entails. There is also a class from Googler Romain Guy that allows you to test a debug version of your app. All of the views and screenshots using the Hierarchy View in the subsequent sections are taken from a Samsung Note II running 4.1.2 Jelly Bean. By testing screen rendering on an older device (with a slower processor), you can be sure that if you meet rendering thresholds on this device, you app will likely render well on all Android devices.

As shown in Figure 4-2, when you open Hierarchy View, there are a number of windows: on the left, there is a Windows tab that lists the Android devices connected to your computer, with a list of all running processes. The active process is displayed in bold. The second tab gives details about a selected build (more on this later). The center section shows a zoomed view of your app's Tree View. Clicking on a view (in this case, the leftmost view) shows you the view as it appears on the device and additional data. To the right are two views: Tree Overview and Layout View. The Tree Overview shows the entire view hierarchy, with a box showing where the zoomed center section is in relation to the entire tree. The Layout View highlights in dark red the area that is painted by the selected view (and light red displays the parent view).

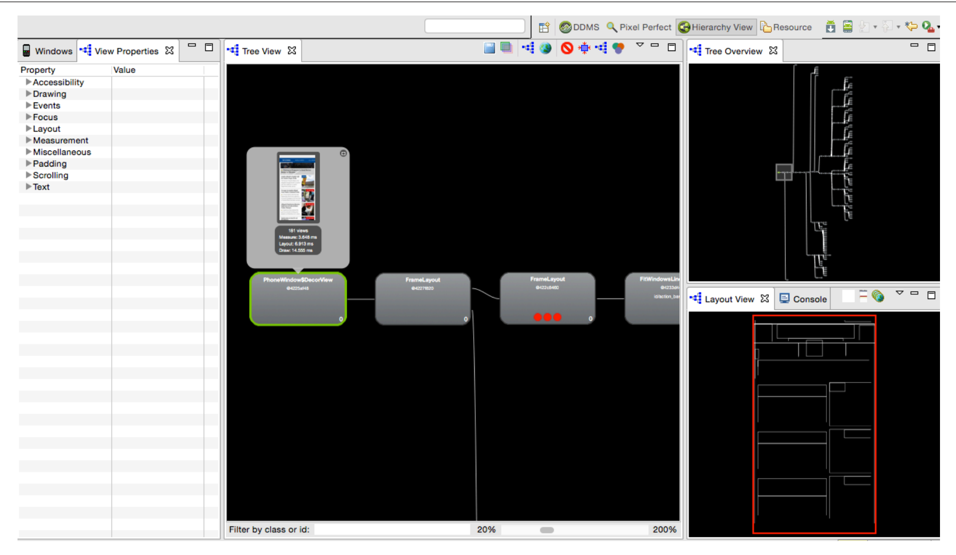


Figure 4-2. Overview of the Hierarchy View tool, using the view tree of a news app

Inside the central close-up view, you can click on an individual view to get a representation of the view on an Android screen. By clicking the green, red, and purple "Venn diagram" icon under the Tree View, this pop-up view will also provide the child view count, and the timing for view measure, layout, and draw. This will calculate the measure, layout, and draw times for every view down the tree from the selection (in Figure 4-3, I chose the top view to obtain the timing to create the entire view).

Figure 4-3. Render times of a view

The topmost view for the article list uses 181 views, measures in 3.6 ms, layout in 7 ms and draws in 14.5 ms (~25 ms total). In order to reduce the time to render these views, it makes sense to look at the Tree Overview of the app, to see how the views fit together as a whole. The tree overview shows that while there are a lot of views in this

screen, the render tree is relatively flat. A flat render tree is good, as the "depth" of your view XML can have detrimental effects to rendering time. Even with the flat XML, there is still a 26 ms draw time, there may be times where this view is janky, and that optimizations should be considered.

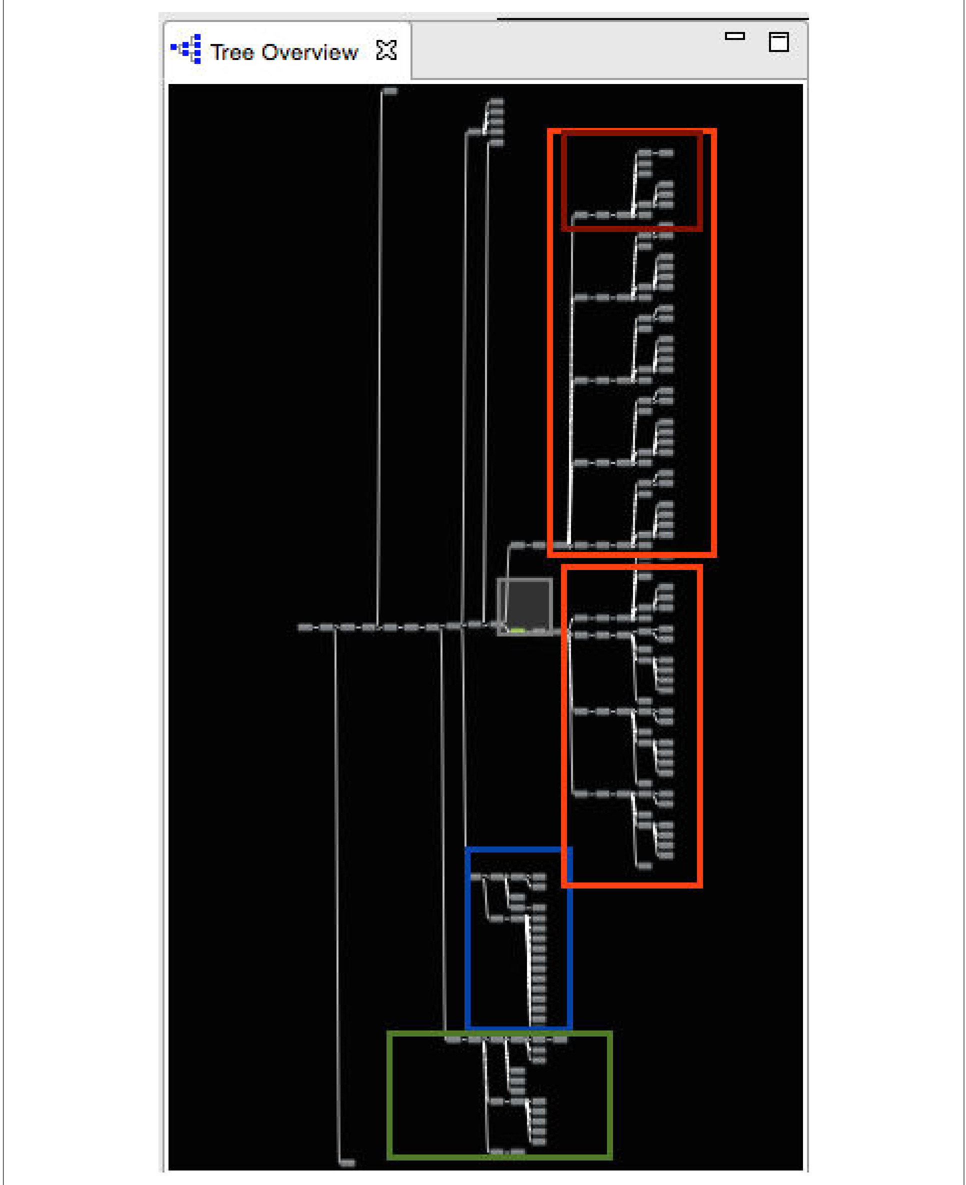


Figure 4-4. Tree Overview

Examining the Tree Overview of a news app's list of articles (Figure 4-4), there are three major regions: the header (in the blue box at the bottom of the views), story lists (there are two orange boxes with views for two different tabs of articles). The views for a single story are highlighted in red. There are nine repeats of this internal headline structure, five in the top orange box, and four in the second). Finally, the views for the side pull-out navigation bar can be found at the bottom (in the green box). The header uses 22 views, the two story lists use 67 and 44, respectively (each headline uses 13 views), and the navigation drawer uses 20. For those of you keeping score, this leaves 18 views unaccounted for. There is a swipe animation, and some interim views that complete the total. As you can see, the number of views can really add up. Being as efficient as possible is crucial to ensuring a jank free experience for your users (see Figure 4-5).

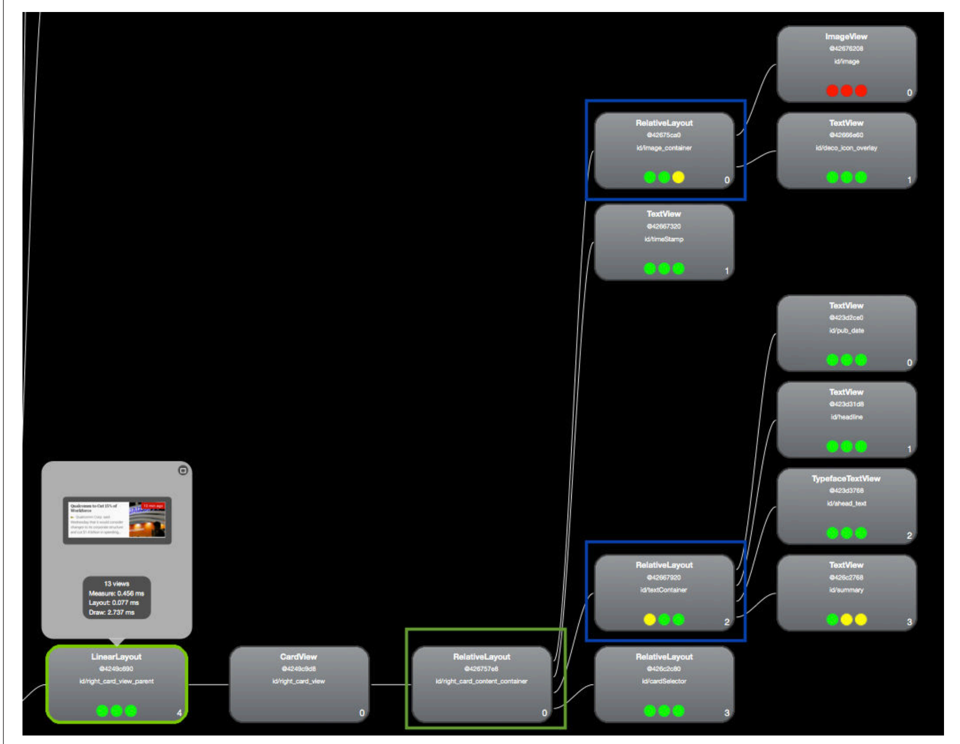


Figure 4-5. Examining a View Tree

Looking closer at a headline, we can look at the 13 views that make up one headline in the list. Each headline has five levels of hierarchy (seen as vertical columns), and it takes .456 ms to measure, 0.077s to layout, and 2.737 to draw. The fifth layer of hierarchy is fed by two RelativeLayouts in the fourth level (highlighted in blue). These are drawn by a third RelativeLayout in the third column (highlighted in green). If the lay-

out described in these two could be described in their parents (in the third level), an entire layer of rendering could be removed. Further, as I explained in Remeasuring Views, each RelativeLayout is measured twice, so having nested RelativeLayouts can quickly lead to increases in measure time.

By now, you may have noticed the red, yellow, and green circles in each view. These denote the relative speed of measure, layout, and draw (from left to right) for that view in that vertical layer of views. Green means fastest 50%, yellow means slowest 50% and red denotes the slowest view in that level of the tree. Obviously, the red views are good places to look for optimizations.

In the tree for the article headline, the slowest view is the ImageView in the upper-right corner. Walking the view back the article parent, it is fed through two Relative-Layouts (increasing the measure time), and then three views with no children (across the bottom). These three views could also be aggregated into a single view—removing two layers of hierarchy.

Let's look at how another news app worked to reduce the number of views per head-line. The Figure 4-6 view shows a similar hierarchy to what we saw in Figure 4-5.

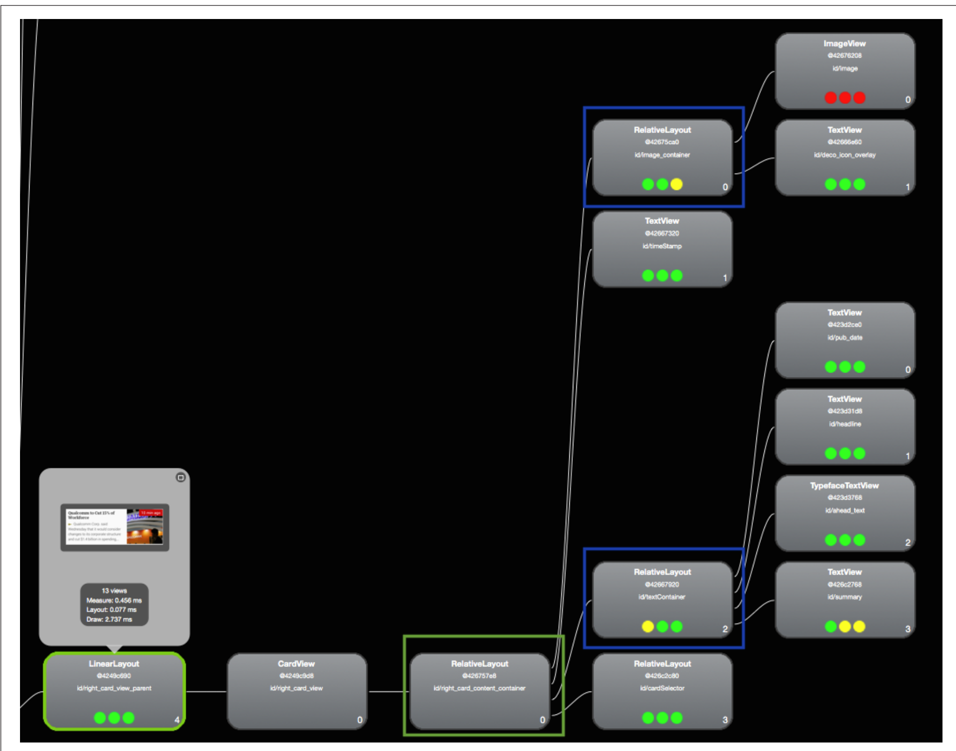


Figure 4-6. Original view tree for a news article

In fact, the headline view (shown in Figure 4-6) has the same issue with RelativeLayouts (in blue), and this results in a measure of 1.275 ms, layout of 0.066 ms and draw of 3.24 ms (a total of 4.6 ms per headline). Upon seeing these times, the developers went back to the drawing board, and built a prettier UI, with a larger image and share buttons—but a flatter hierarchy (Figure 4-7).

Figure 4-7. Updated View Tree

Now the headline (with just three columns of hierarchy) takes a total of just 4.2 ms to render—a savings of 400 ms even with the larger UI!

To better explore aspects of performance, I will use examples from a sample app, "Is it a goat?" This simple app is a list of several images with checkmarks next to pictures of goats. This sample app has several different layouts built into it, from unoptimized and slow to an optimized fast XML layout. By examining the views, and how they evolve, we can quantify how the optimizations improve the rendering of the app. We'll walk through several steps of optimization in this app, and each change in view layout can be viewed in Hierarchy View by changing the view in the settings. Upon

choosing a layout type, the view is refreshed with a more (or less) optimized XML view structure. We'll start with the "Slow XML" as the unoptimized starting point. A quick look at the Hierarchy View at the unoptimized version of this app reveals a few things, as shown in Figure 4-8.

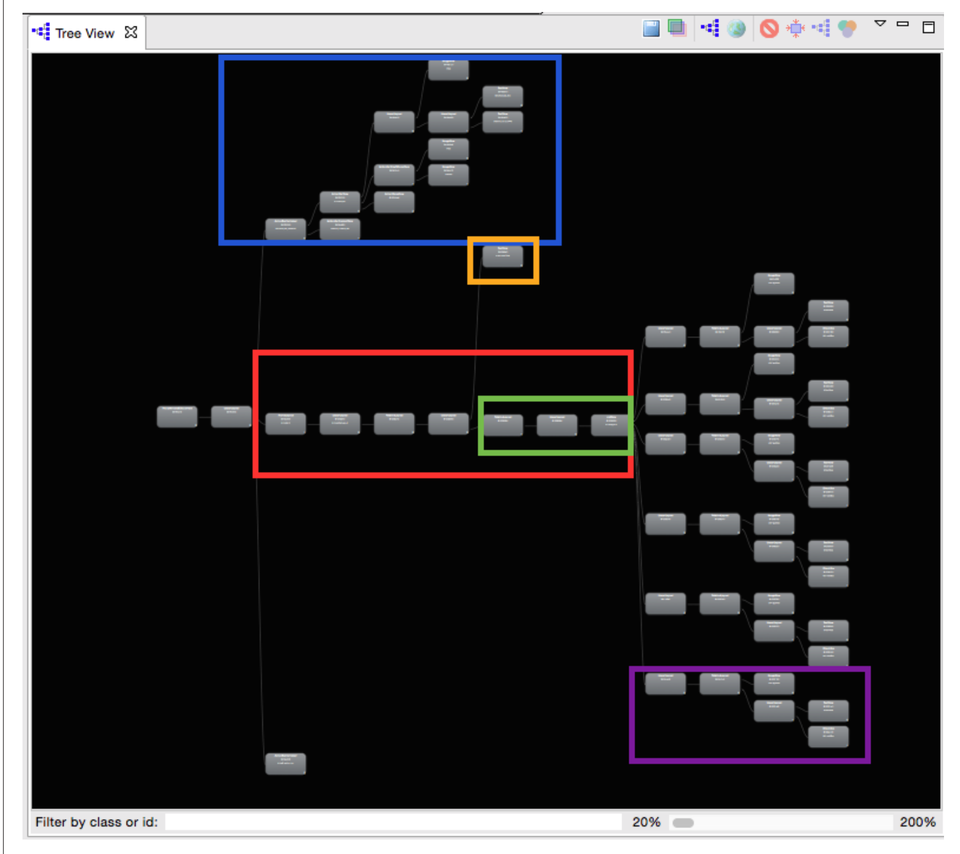


Figure 4-8. Hierarchy view of the unoptimized "Is it a Goat?" app

There are 59 views in this simple app. However, unlike the news app in Figure 4-4, the view tree has more horizontal depth. The more views that are fed on top of one another, the longer the app will take to draw. By removing layers of depth, this app will render each frame on the screen faster.

The blue box frames out the views for the Android Action Bar. The orange box is the text box at the top of the screen, and the purple box marks one row of goat information (there are six identical views above the purple view indicated). The red box shows seven views in a row that only add depth to the app, and do not build any additional display. Taking a closer look at just three of these sequential views (in the green box) shows an interesting remeasurement issue (Figure 4-9).

Figure 4-9. Remeasurement in Hierarchy View

As the device measures views, it starts from the right (child view) and moves to the left (to the parent views). The ListView on the right takes the measurements from the six rows of goat data (37 total views), and takes 0.012 ms to measure. This feeds into the center LinearLayout (38 views). Interestingly, the measure timing balloons out due to a remeasurement loop. The measure time jumps three orders of magnitude to 18.109 ms. The RelativeLayout to the left of the LinearLayout redoubles the measurement time to 33.739 ms. By the time the measurement cascades through the additional parent views (those additional views in the red box in Figure 4-8), the total measurement time is over 68 ms. By simply removing this one LinearLayout, the entire view tree measurement drops to under 1 ms! (You can verify this by comparing the "Remove Overdraw" layout in the app to the "Remove LL+OD" layout. The only difference is removing that one view.) We can remove several more layers of depth by applying the "Optimized Layout" setting. From the seven excess views shown in Figure 4-10, the app now has just three layers (Figure 4-11).

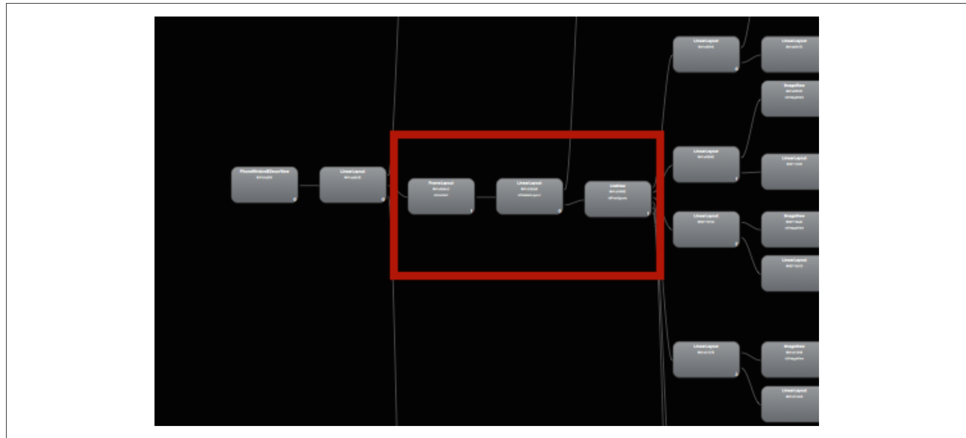


Figure 4-10. Removing hierarchy depth

A further optimization to remove view depth can be done by looking at the rows of goat data. Each line of goat information has six views, and there are six rows of data visible on the screen (one such row is highlighted in a purple box at the bottom right of Figure 4-8.) Using the Hierarchy View tool to look at how the views are built for one row of the Goat app (Figure 4-11), we see that the two left-most views (a LinearLayout and RelativeLayout) add only depth to the views ("Slow XML" view). The initial LinearLayout feeds directly into a RelativeLayout, but adds nothing to the display.

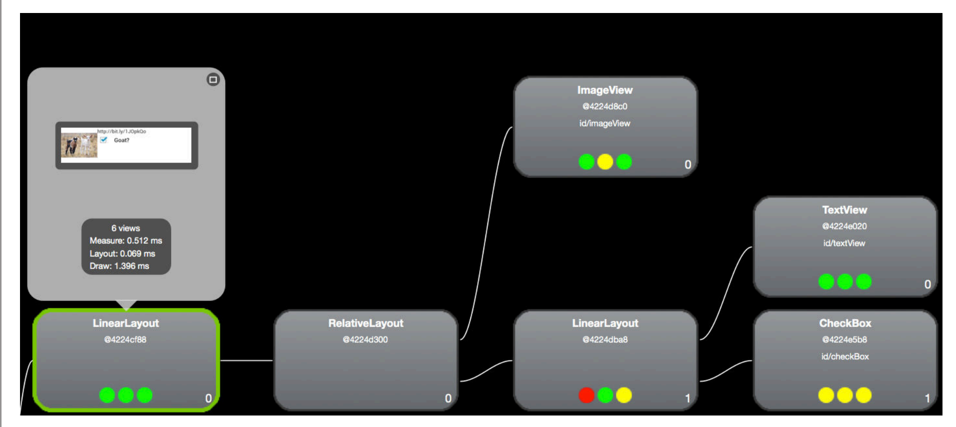


Figure 4-11. Unoptimized hierarchy view of goat row

Because RelativeLayouts remeasure twice (and we are trying to reduce measurement time), I first attempted to remove the RelativeLayout ("Optimized Layout" setting in the app; see Figure 4-12). When I did this, the depth was reduced from 4 to 3, and the display rendering is faster.

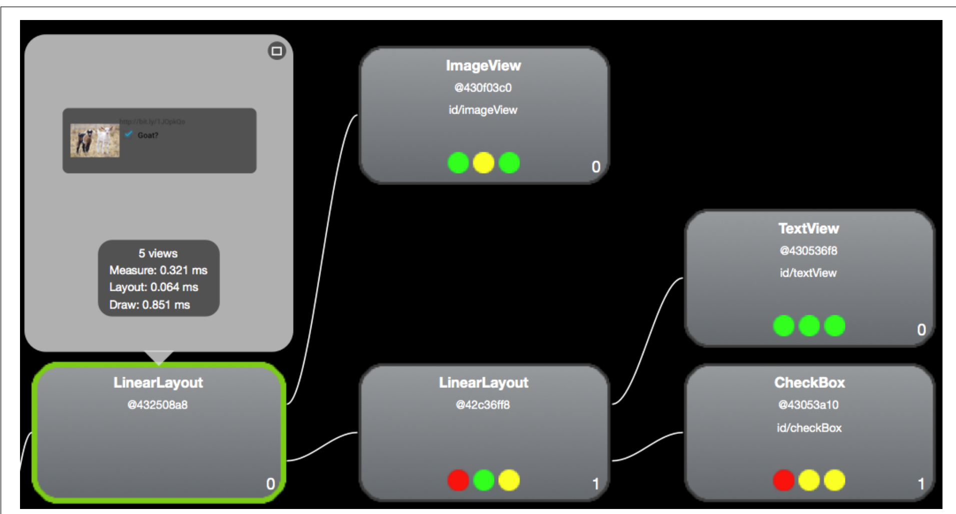


Figure 4-12. Views optimized by removing RelativeLayout

However, this is not the fastest optimization. By removing the LinearLayout and reorganizing the RelativeLayout to handle the entire row of information (as shown in Figure 4-13), the view depth is reduced to 2. The layout is 0.1 ms faster to render. It just goes to show that there is more than one way to optimize your layouts, and it does not hurt to test different options (see Table 4-1).

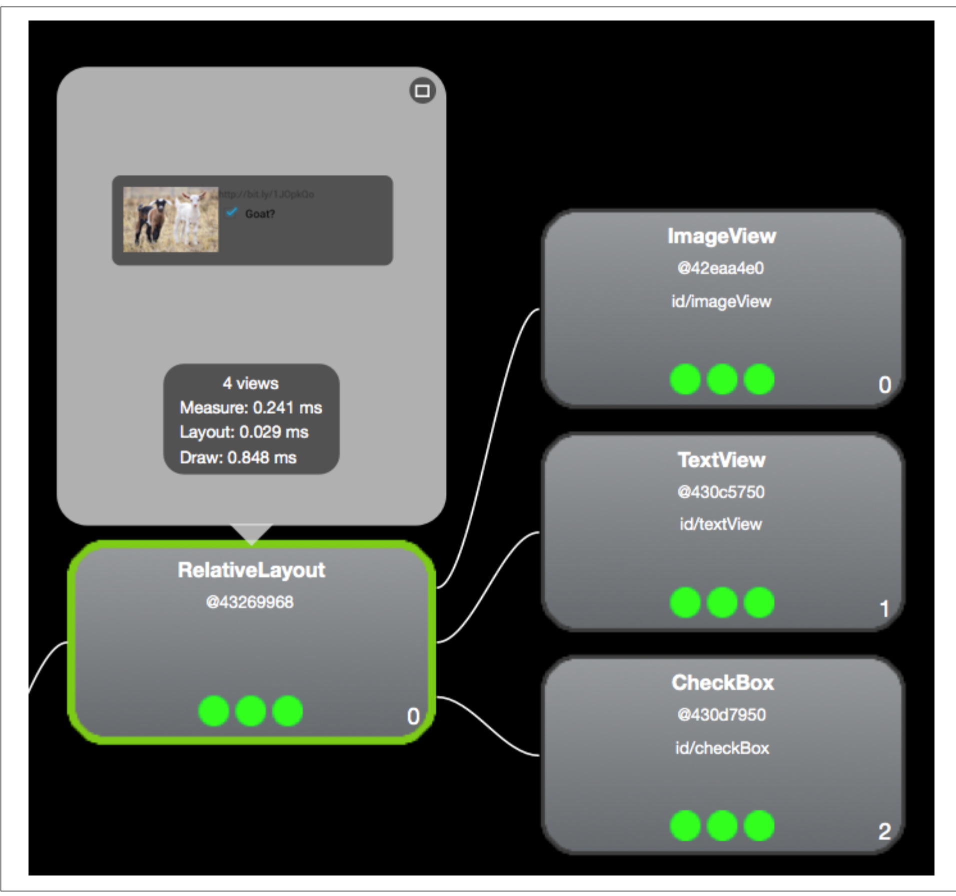


Figure 4-13. Views optimized by consolidating into RelativeLayout

Table 4-1. View Tree optimization improvements

Version	View Count	View Depth	Measure	Layout	Draw	Total
Unoptimized	6	4	0.570	0.068	1.477	2.115 ms
Remove RelativeLayout	5	3	0.321	0.064	0.851	1.236 ms
Remove LinearLayouts	4	2	0.241	0.029	0.848	1.118 ms

By removing ~1 ms of rendering from each row of information, we can pull about 6 ms from the entire render time (assuming six rows of data on the screen). If your app has jank, or your tests show that you are close to the 16 ms border of jank, saving 6 ms will definitely pull you further from the edge.

Reusing Views

Like a good object-oriented programmer, you likely have views that you call and reuse (rather than recode over and over). In my “Is it a goat?” app, the goatrow layout is reused for every row of data. If the view wrapping your sublayout is only used to create the XML file, it may be creating an extra layer of depth in your app. If this is the case, you can remove that outer view wrapper and enclose the elements in `<merge>` tags. This will remove the extra layer of hierarchy from your app.

As an exercise, download the “Is it a goat?” app on GitHub and observe the view times in the Hierarchy View tool. You can modify the view XML files used by changing the radio buttons in the settings menu, and use the Hierarchy View tool to view the changes in depth of the app, and how these changes affect the rendering speed of your app.

Hierarchy Viewer (Beyond the Tree)

The Hierarchy Viewer has a couple of additional neat functions that can be helpful to better understand overdraw. Following the options in the tree view from left to right, you have the ability to do a number of useful things like:

- Save any view from the tree as a PNG (icon is a stylized diskette).
- Photoshop export (described in “Overdrawing the Screen” on page 90).
- Reload the view (second purple tree icon).
- Open large view in another window (the globe icon); this has an option to change the background color so that it’s easier to determine whether there is overdraw.
- Invalidate the view (red line with bar through it).
- Request view to layout.
- Request view to output the draw commands to the LogCat (yes, the third use of the purple tree icon); this is a great way to read the actual OpenGL commands for each action being taken. For Open GL experts: this will be helpful for in-depth optimizations.

It is clear that the Hierarchy Viewer is a must-have analysis tool to optimize the View tree of your app—potentially shaving tens of ms from the render time of your Android app.

Asset Reduction

Once your app is flattened and the number of views are reduced, you can also reduce the number of objects used in each view. In 2014, Instagram reduced the number of assets in its title bar from 29 to 8 objects. They quantified the performance increase to be 10%–20% of the startup time (depending on device). They managed this reduction through asset tinting, where they load just one object, and modify its color using a ColorFilter at runtime. For example, by sending your drawable and desired color through the following method:

```
public Drawable colorDrawable(Resources res,
  @DrawableRes int drawableResId, @ColorRes int colorResId) {
    Drawable drawable = res.getDrawable(drawableResId);
    int color = res.getColor(colorResId);
    drawable.setColorFilter(color, PorterDuff.Mode.SRC_IN);
    return drawable;
}
```

One file can be used to represent several different object states (starred versus unstarred, online versus offline, etc.).

Overdrawing the Screen

Every few years, there is a story about how a museum has X-rayed a priceless painting and discovered that the artist had reused the canvas, and that there was an undiscovered new painting underneath the original masterwork. In some cases, they are even able to use advanced imaging techniques to discover what the original work on the canvas looked like. Android views are drawn in a similar manner. When Android draws the screen, it draws the parent first, and then the children/grandchildren/etc. views on top of the parent views. This can result in entire views being drawn on the screen, and then—much like the artist and his canvas—these views are entirely covered up by subsequent views.

During the Renaissance, our master artist had to wait for the paint to dry before he could reuse his canvas. On our high-tech touch screens, the speed of redrawing the screen is several orders of magnitude faster, but the act of painting the screen multiple times does add latency, and potentially can add jank to your layout. The act of repainting the screen is called overdraw, and we'll look at how to diagnose overdraw in the next section.

An additional problem with overdraw is that anytime a view is invalidated (which happens whenever there is an update to the view), the pixels for that view need to be redrawn. Because Android does not know which view is visible, it must redraw every view that is associated with those pixels. In the painting analogy, our artist would have to scratch out all the paint back to the canvas, repaint the "hidden masterwork" and then repaint his current work. If your app has multiple layers or views being

drawn for that pixel, each must be redrawn. If we are not careful, all of this heavy lifting to draw (and redraw) the screen can cause performance issues.

Testing Overdraw

There are a number of great tools offered from Android to test overdraw. In Jelly Bean 4.2, the Debug GPU Overdraw tool was added in the Developer Options menu. If you are using a Jelly Bean 4.3 or KitKat device, there is a version of the Overdraw counter that gives you a weighted average of total screen overdraw in the bottom left of the view. I find that this tool is a very useful way to quickly look at apps for overdraw issues. However, it does appear to overestimate apps that have more than 6–7x overdraw (yes, it happens more than we'd like to admit).

The screenshots shown in Figure 4-14 are again from the "Is it a goat?" app. The overdraw counters can be seen in the lower left. There are three overdraw counters on the screen, but the one we can control as a developer appears in the main window. The overdraw counter appears in the bottom left. The unoptimized app on the left has an overdraw of 8.43, and our optimization steps will reduce this to 1.38. We can also see that the nav bars have overdraws of 1.2 (and the menu buttons 2.4), meaning that the text and icons overdraw this section by an extra 20% (140%). While the overdraw counter is a quick way to compare overdraw between apps without impacting the user experience too much, it does not help you understand where the overdraw issues lie.

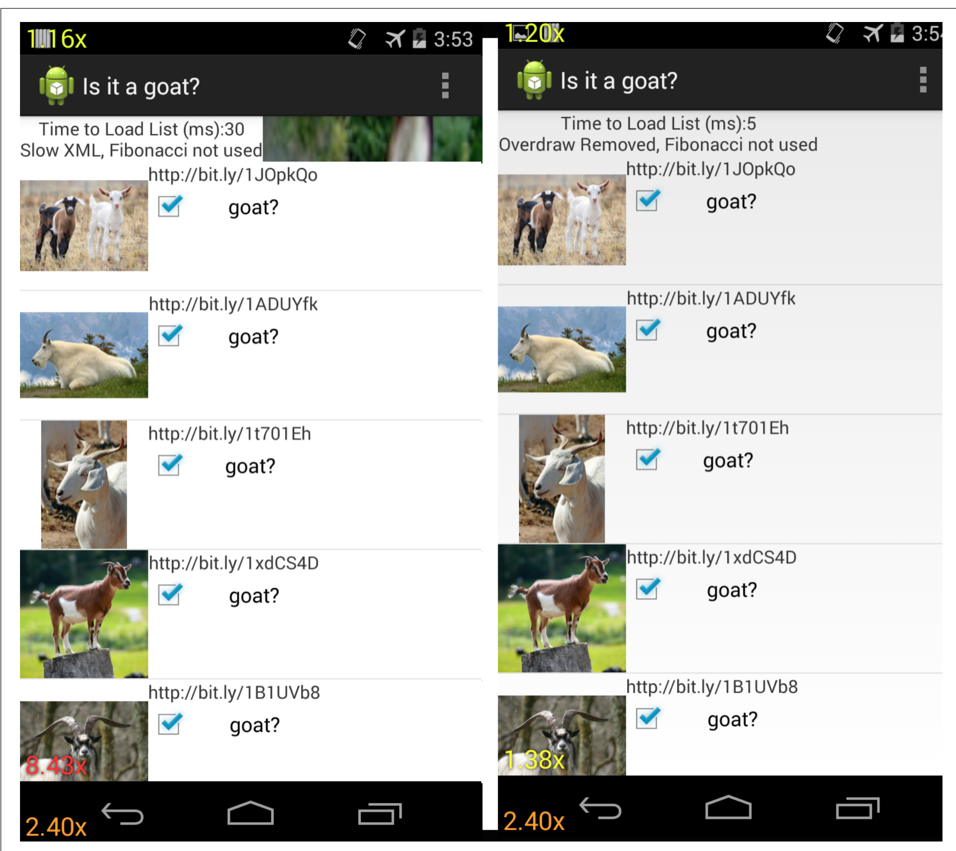


Figure 4-14. Overdraw counter for unoptimized (left) and optimized (right) views of the same app

Another way to visualize the overdraw is to use the “Show Overdraw areas” selection in the Debug GPU overdraw menu. This tool makes places an overlay of color over your app, showing you the total amount of overdraw in each region of the app (for those developers who are colorblind, the KitKat release offers a colorblind-friendly setting). By comparing the colors on the screen, you can quickly determine the issues at hand:

White
: No overdraw

Blue
: 1x overdraw (screen is painted twice)

Green
: 2x overdraw (screen is painted twice)

Light red
: 3x overdraw

Dark red
: 4x or more overdraw

In Figure 4-15, you can see the overdraw areas rendering of the "Is it a goat?" app before and after optimization. The menu bar of the app is not colored (no overdraw) in either screenshot, but the Android icon and the settings menu icon are green (2x overdraw). The list of goat images is dark red before optimization (indicating at least 4x overdraw). After the app views were optimized, there is now only blue (1x) overdraw over the checkbox and the images—indicating that at least three layers of drawing were removed! There is now no overdraw around the text and in the *blank space.*

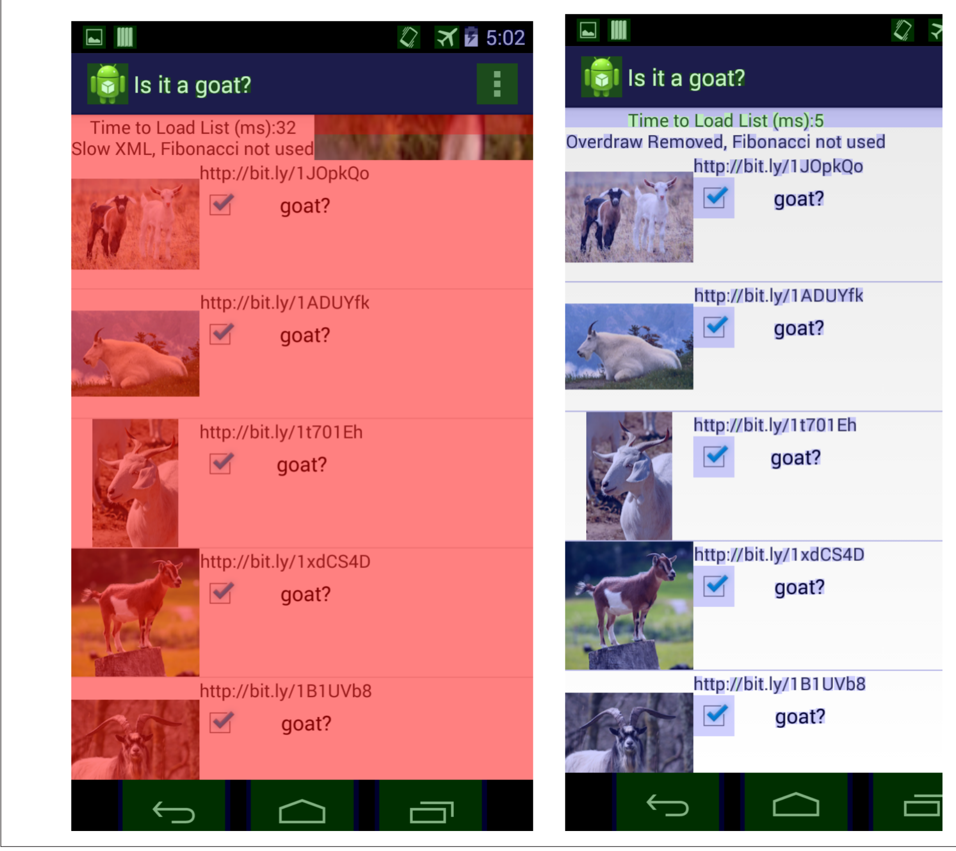


Figure 4-15. Overdraw colors before optimization (left) and after (right)

By reducing the number of views (or at least the way these views over paint one another) the app will render faster. Comparing the parent view in the Hierarchy Viewer for the view with excess overdraw and the optimized version ("Slow XML" versus "Remove Overdraw") shows a 50% drop in the draw time from 13.5 ms to 6.8 ms.

Overdraw in Hierarchy Viewer

Another way to visualize the overdraw in an app is to save the view hierarchy as a Photoshop document (the second option on the Tree View) in Hierarchy Viewer. If you do not have Photoshop, there are a number of free tools available that will allow you to open this document (the subsequent screenshots are from GIMP for Mac). When opening these views, you can really see the overdraw present in different layers. In most production apps, it is typically drawing a white background on top of another white background. This does not sound terrible, but it is two steps of painting, and should be avoided. To better visualize this in the "Is it a goat?" app, all overdrawn regions utilize an image of a donkey instead of a white background. If you look at the images in previous pages, there are no images of a donkey visible, because they were overdrawn with a white view on top of them. By removing the visible view layers, we'll be able to see the layers of donkey below, and quickly determine where overdraw occurs, and then remove it. In GIMP, views that are visible in your app have a small eye icon next to the layer. In Figure 4-16, you can see that I have begun to peel back the views at the top of the "Is it a goat?" app (revealing a large donkey). In the layout view list to the right, you can see there are a number of full screen layouts that are visible (and they all are showing the same donkey image).

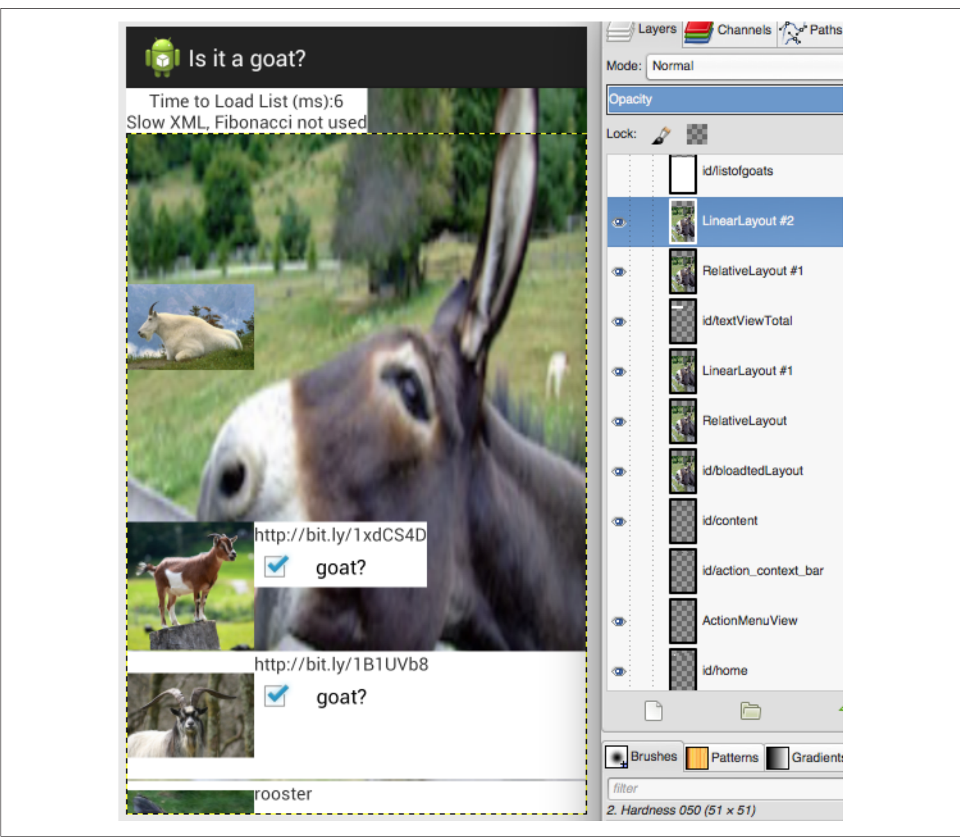


Figure 4-16. Visually peeling back views

Another way to visualize the "peeling back of the views" is shown in Figure 4-17. We start at the top left with the full screen view of the app, as seen on the device. Moving to the center top screenshot, we have removed two rows of goat pictures and layout, revealing that under each row of goat data there is a stretched picture of a donkey. Below the six or seven stretched small donkey images, there is a white backdrop (seen in the rightmost image on the top row with two of the small donkey pictures). Removing that white layer reveals a large donkey, as seen at the bottom left. Below the donkey pictures, there is a final full screen of white until we reach the bottom of the view tree (seen at the bottom right).

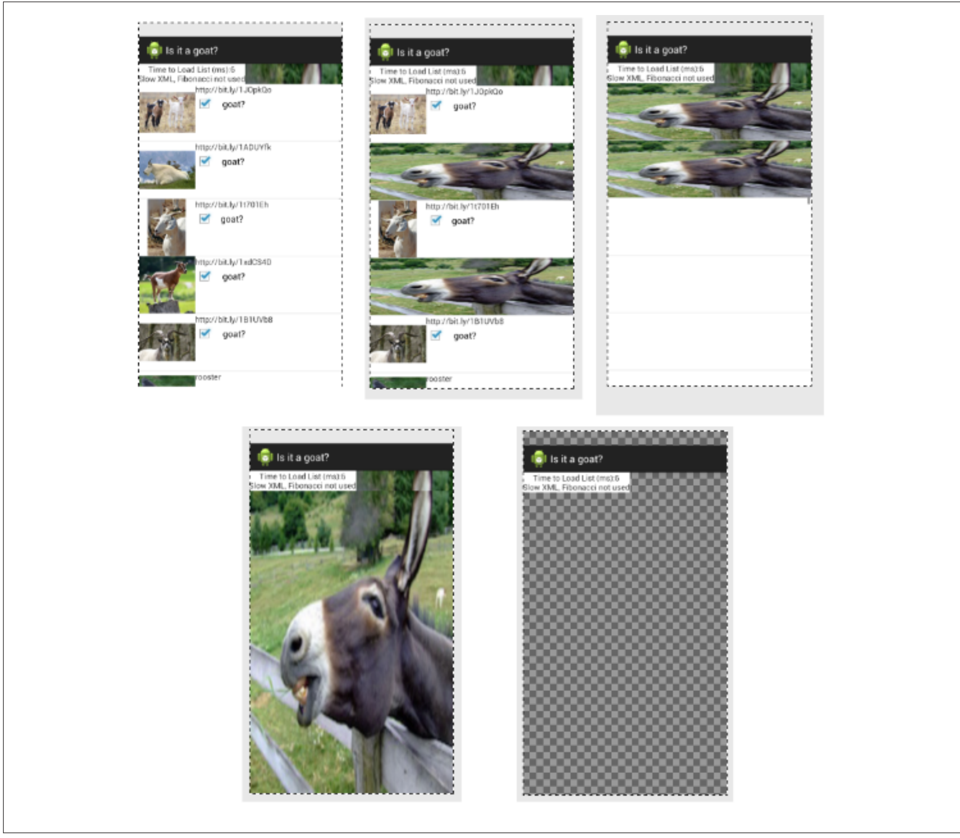


Figure 4-17. Looking at the layers visually

Overdraw and KitKat (Overdraw Avoidance)

In KitKat and newer devices, the effects of overdraw have been dramatically reduced. Called Overdraw Avoidance, the system can remove simple cases of overdraw (such as views that are completely covered by other views) automatically (this likely means that the effects of the full screen donkeys in my “Is it a goat?” app will not be felt by KitKat and newer users). This will improve the draw rate for apps with overdraw, but it still makes sense to clean up as many examples of overdraw as possible (for better code, and for your customers on Jelly Bean and lower).

Overdraw Avoidance and Developer Tools

When you use the Overdraw tools described earlier, KitKat’s Overdraw Avoidance is disabled, so you can see what your layout really looks like, but not how the device actually sees it.

Analyzing For Jank (Profiling GPU Render)

After the view hierarchy and overdraw have been optimized, you may still be suffering from lost frames or choppy scrolling: your app still suffers from a case of jank. You may not experience jank on your high-end Android device, but it might be there on the devices with less computing power. To get an overall view of the jank in your app, Android has added Profile GPU Rendering as a Developer Option in Jelly Bean and newer devices. This measures how long it takes each frame to draw onto the screen. You can either save the data into a logfile (`adb shell dumpsys gfxinfo`), or you can display the GPU rendering as a screen overlay in real time on the device (available on Android 4.2+).

For a quick analysis of what is going on, I really like displaying the GPU rendering on the screen to get a holistic view of what is happening, (but the raw data from the log is great for offline graphing or reporting). Again, this is good to attempt on multiple devices. In Figure 4-18, you can see the GPU rendering profile on a Nexus 6 running Lollipop (left) and the Moto G on KitKat (right) for the "Is it a goat?" app. The bars appear at the bottom of the screen. The most important feature in this GPU profile graph is the horizontal green bar. This frame denotes the 16 ms time the device uses to render a frame. Each frame that is rendered is a horizontal bar. If you have a lot of frames that go over the 16 ms line, you have a jank problem. In the figures below, there are a few instances of Jank on the Nexus 6. This occurred when the scrolling hit the end of the page, and the device did a bounce animation. The end-user experience was not terribly affected. Each screen draw (vertical line) is broken down into four additional measurements collected (on Lollipop) by color: draw (blue), prepare (purple), process (red), and execute (yellow). In KitKat and earlier, the prepare data is not broken out, and is included in the other metrics (hence only three colors appear in the KitKat GPU profile screenshots).

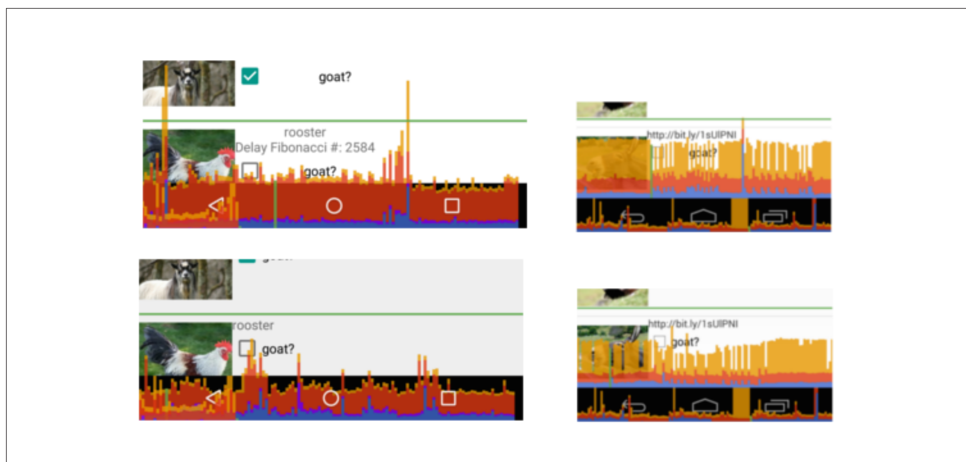


Figure 4-18. GPU Profiling Lollipop (left) and KitKat (right) of the unoptimized “Is it a goat?” app (top), optimized (bottom)

Comparing the GPU data from the Nexus 6 to the Moto G brings us back to the topic of device testing. The unoptimized “Is it a goat?” app (top row) in Figure 4-18 qualitatively shows that the Moto G takes twice as long as the Nexus 6 (by comparing vertical heights of the GPU profile to the green line, scales are the same). This can be quantified by collecting the data (`adb shell dumpsys gfxinfo`) and graphing. In the next example, the optimized view takes almost twice as long on the Moto G. For both devices, the draw, prepare, and process steps all take about the same amount of time (less than 4 ms total). The difference occurs in the execute phase (purple) of the frame draw, where the Moto G often takes ~4 ms longer than the Nexus 6. This goes to show that testing GPU rendering is best done on your lower-powered devices, as they are more likely to have issues rendering your views without jank.

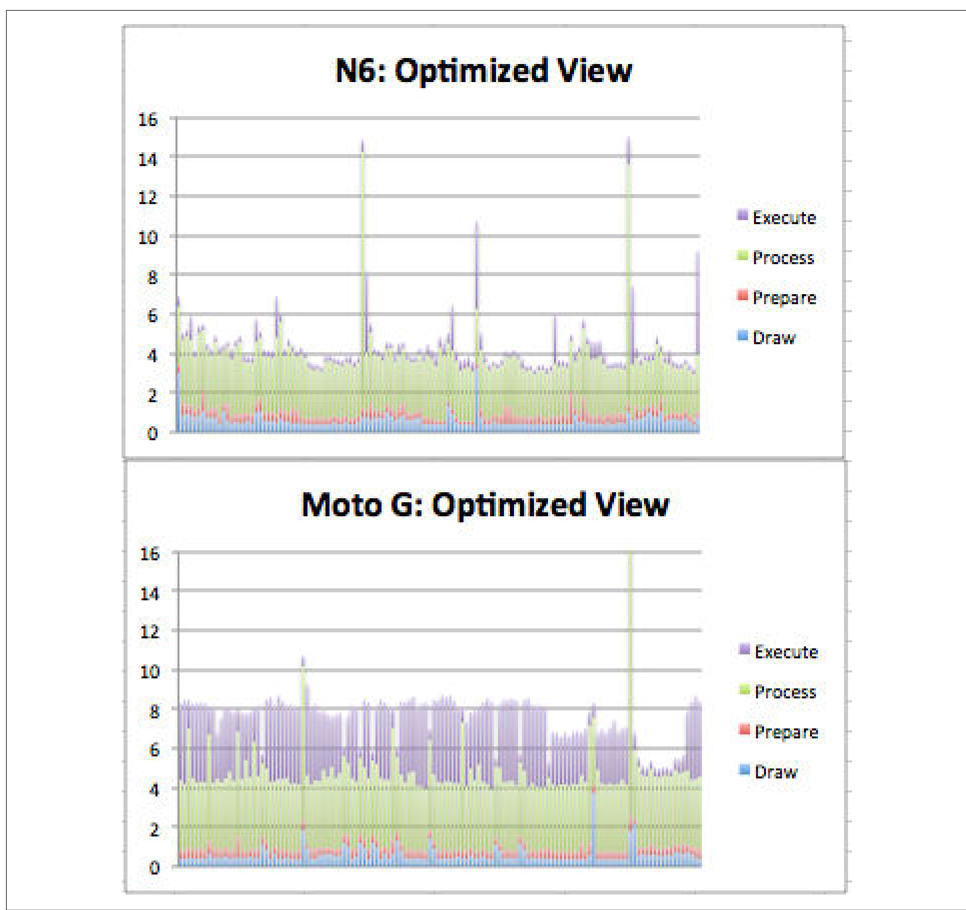


Figure 4-19. GPU Profiling Lollipop (top) and KitKat (bottom) of the optimized views

At a high level, the GPU profiler lets you know you might have a problem. In the "Is it a goat?" app, if I turn on the Fibonacci delay (where a heavy recursive calculation is done during view creation), the GPU profiler does not show any jank because the calculation takes place on the UI thread and completely blocks rendering (on slower devices this setting results in an app not responding message).

Fibonacci Calculation Algorithms

The Fibonacci sequence is a series of numbers where each value is the sum of the two preceding values: 0, 1, 1, 2, 3, 5, 8, and so on. It is commonly used to describe recursion in programming, and in this case, I am using the most inefficient code to generate the Fibonacci value:

```
public class fibonacci  {
        //recursive Fibonacci
        public static int fib(int n){
            if (n<=0)
                return 0;
            if (n==1)
                return 1;
            return fib(n-1) + fib(n-2);
        }
```

The number of calculations required to generate each value grows exponentially. The goal here is to put so much work on the CPU during rendering that the views are delayed and cannot render quickly. Calculating n=40 really slows down the app (and causes it to crash on lower-end devices). While perhaps a slightly contrived example of what might block your views from rendering, the techniques we used to identify the Fibonacci code in our traces will help you find code that is slowing down your app.

GPU Rendering in Android Marshmallow

In Android Marshmallow, `adb shell dumpsys gfxinfo` *<packagename>* adds several new features to aid in your quest for jank free rendering. First off, the report now leads off with a summary of every frame rendered by your app:

```
** Graphics info for pid 2612 [appname] **

Stats since: 1914100487809ns
Total frames rendered: 26400
Janky frames: 5125 (19.41%)
90th percentile: 20ms
95th percentile: 32ms
99th percentile: 36ms
Number Missed Vsync: 142
Number High input latency: 11
Number Slow UI thread: 2196
Number Slow bitmap uploads: 439
Number Slow draw: 3744
```

From the time the app was started, now you can see how many frames were rendered, and how many are 90th percentile and the timings for the slowest frames (90th, 95th, and 99th percentile). The last five lines list reasons that the frame did not render in 16

ms. Note that there are more issues than janky frames, indicating that some frames were impacted by more than one issue.

Another great addition to Android Marshmallow to the gfxinfo library of test tools is `adb shell dumpsys gfxinfo <packagename>` framestats. This outputs a large comma-separated table with specific timings of events in each frame. The columns in the export are not labeled, but are described at the Android developer site. To determine the time each step of the rendering pathway takes on your device, you must calculate the differences between the framestats reported values. To simplify these calculations, I have created a spreadsheet that computes the values of interest. When you paste in the raw CSV data, columns P-X become populated with useful data about each frame render (all results are in ms):

- VSYNC-Intended_VSYC (tells you if a frame render was missed—jank!)
- Input event time (processing time for input events—should be < 2 ms))
- Animation evaluation (should be < 2 ms)
- Layout and measure
- view.draw() time
- Sync phase time (if > 0.4 ms, indicates many new bitmaps being sent to GPU)
- GPU work time (overdraw draw time will appear here)
- Total frame time

There are two tabs in the worksheet with sample data, both from the "Is it a goat?" app: goat-optim and goat-slowXML. Looking at the data from the goat-slowXML sheet (shown in Figure 4-20), we can see a few frames (in purple) where the total frame draw exceeded 16.6 ms. Fortunately, due to the presence of frames in the VSYNC buffer, no frames were dropped (as indicated by the 0s in the first column). For devices with a smaller buffer (or for apps where the buffer does not have time to repopulate), this could result in a janky user experience. The chart also implies that slow input events (orange) and evaluate animator events (red) add GPU work, and lengthen the total frame rendering time.

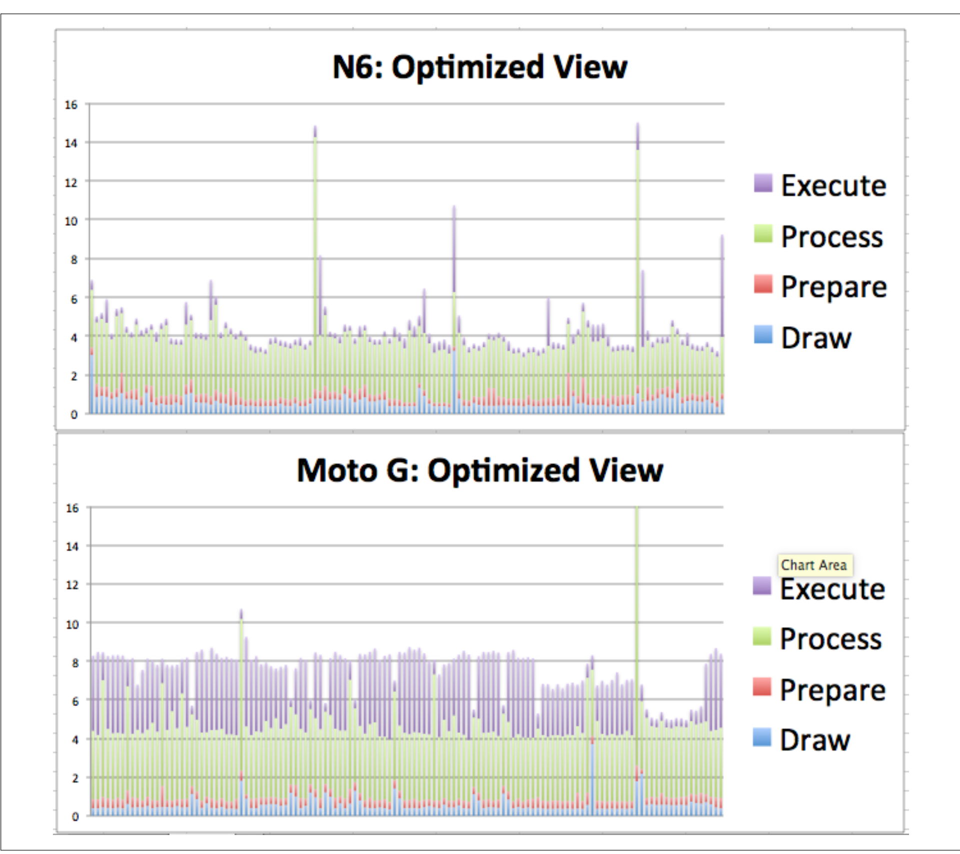


Figure 4-20. Data from gfxinfo framestats

Beyond Jank (Skipped Frames)

There are times that the GPU profiler does not show a jank event crossing the 16 ms threshold, but you can tell that there was a skip or jump in the UI rendering. This can occur during a skipped frame, where rendering is completely blocked by the CPU doing something else. In Monitor or Android Studio, you can watch the logfiles in the DDMS view. It is easier follow logs from your app if you filter on the process you are testing. If you think this might be the case in your app, look in the logfiles for a warning like the one shown in Figure 4-21.

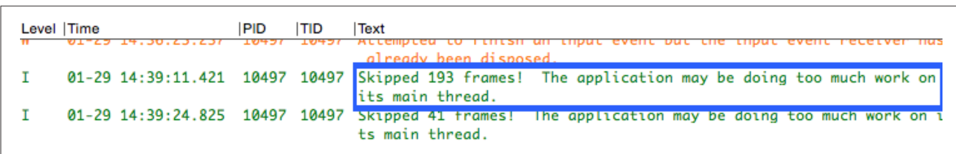


Figure 4-21. Log error showing skipped frames

We'll look at how skipped frames are caused by the CPU in Chapter 5.

Systrace

If you are still experiencing jank after optimizing all of your views, all is not lost. The Systrace tool is another way to measure the performance of your app, and it can also help you diagnose where the issue might lay. Introduced as a part of "Project Butter" with the Jelly Bean release, it allows quick scans into how your app is behaving at core levels of your Android device. There are many types of Systrace parameters that can be run, and we will cover other traces later in the book. Here we will focus on how the UI is rendered, and debug issues associated with jank using Systrace. All of the traces shown in this chapter are available in in the *High Performance Android Apps* GitHub repository.

Systrace differs from the previous tools in this chapter in that it records data for the entire Android system; it is not app specific. For this reason, it is best run on a device that has few additional processes running at the same time, so that the other processes do not interfere with your debugging. In the examples here, we Run Systrace from Android Monitor (but it can also be run from Android Studio or the command line). The Systrace icon is a stylized green and pink graph icon (marked by a red oval in Figure 4-22), and when you press it, it opens a window with a number of options.

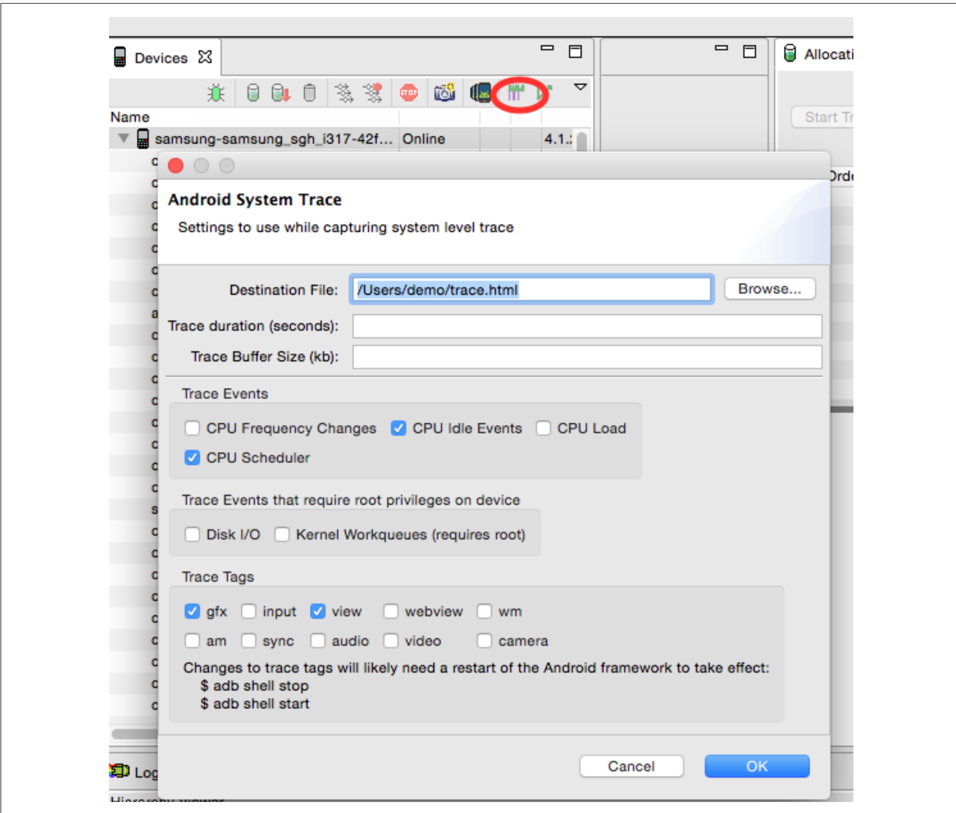


Figure 4-22. Starting with Systrace

The trace is recorded into an HTML file that can be opened in your browser. To study the interactions on the screen, we'll just collect the CPU, graphics and view data (as shown in the dialog box in Figure 4-22). We'll leave the duration field blank (default to 5 seconds). When you press OK, the Systrace will immediately begin to record the parameters you selected on the device (so you'd better be ready to start right away). Because the trace is extremely detailed (and measures all features to the sub-millisecond timeframe), it should be used to diagnose one issue at a time rather than to get a holistic view of your app's performance.

Much like in "Battery Historian" on page 52 in Chapter 3, the output from these traces is overwhelming (and we only picked four of the options available!). Scrolling can be performed with the mouse, and the WASD keys are used to zoom in/out (W, S) and scroll left/right (A,D). At the top of the trace just run, you'll see details about the CPUs. Below the CPU data are collapsible sections describing each process that was active. Each color bar indicates a different action by the OS, and the length of the color indicates the duration (and if we zoom in, we'd see even more lines). Selecting a

bar provides details about that item in the window at the bottom of the screen. Like Battery Historian and other tools, the high-level view (shown in Figure 4-23) is intimidating at first glance. Let's take our first look, and then dig into the information provided so that you can become an expert at reading these files.

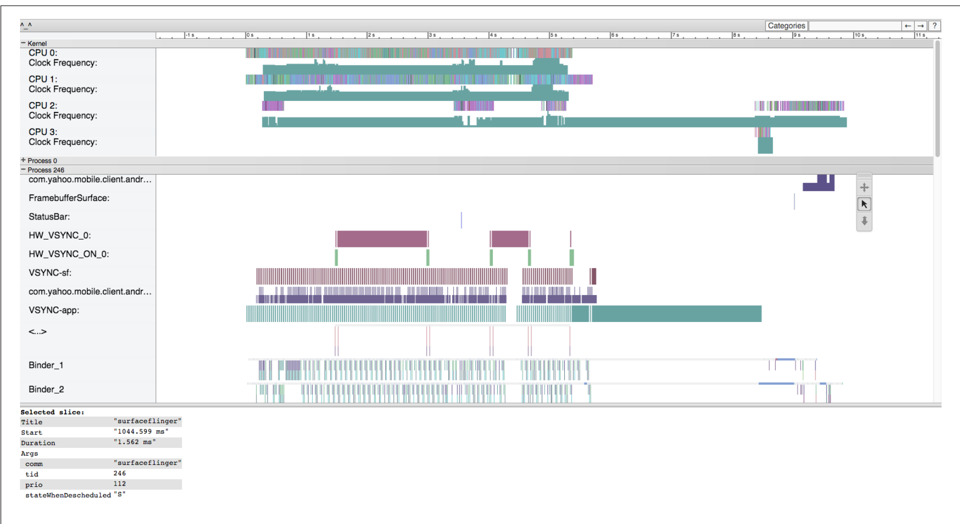


Figure 4-23. Starting with Systrace (Lollipop)

Systrace Evolution

Like the Android ecosystem itself, Systrace has a slightly different interface, display, and set of results depending on the version of the OS you are testing:

- On Jelly Bean devices, there is a setting in Developer Options to enable tracing. You must enable the trace collection on both the computer and the device.
- The output from each release of Android becomes more detailed and has slightly different layouts.
- It is still worthwhile to look at Systraces from Jelly Bean and compare to Lollipop, as you can glean different information from the devices, but they will look different.

At Google I/O 2015, a new version of Systrace was launched, and some of the new features are discussed in "Systrace Update—I/O 2015" on page 115.

As we scroll down through the Systrace results, every process that ran during the test can be seen. For the study of jank, we are primarily looking at the way the app in

question draws, and the when the screen refreshes. As long as these two partners are in sync, the dance of the screen rendering will be smooth. However, should either take a misstep, there will be the opportunity for there to be a jitter or jank in the rendering of the page.

Systrace Screen Painting

Let's walk through the steps of screen painting, using Figure 4-24 as an example. The top row of the trace (highlighted in blue) is the VSYNC, consisting of wide, evenly spaced teal bars. VSYNC is the signal to the OS that it is time to update the screen. Each bar denotes 16 ms (as does the whitespace between the bars). When a VSYNC event occurs (at either end of the teal bar), the surfaceflinger (highlighted with a red box and consisting of several colors of bars from purple-orange and teal) to grab a view from the view buffer (not shown) and displays the image on the screen. Ideally, surfaceflinger events will be 16 ms apart (no jank), so gaps indicate times where the surfaceflinger missed a VSYNC update—the screen did not update in time (and where to look for causes of jank). You can see such a gap about 2/3 of the way through the trace (highlighted in a green box).

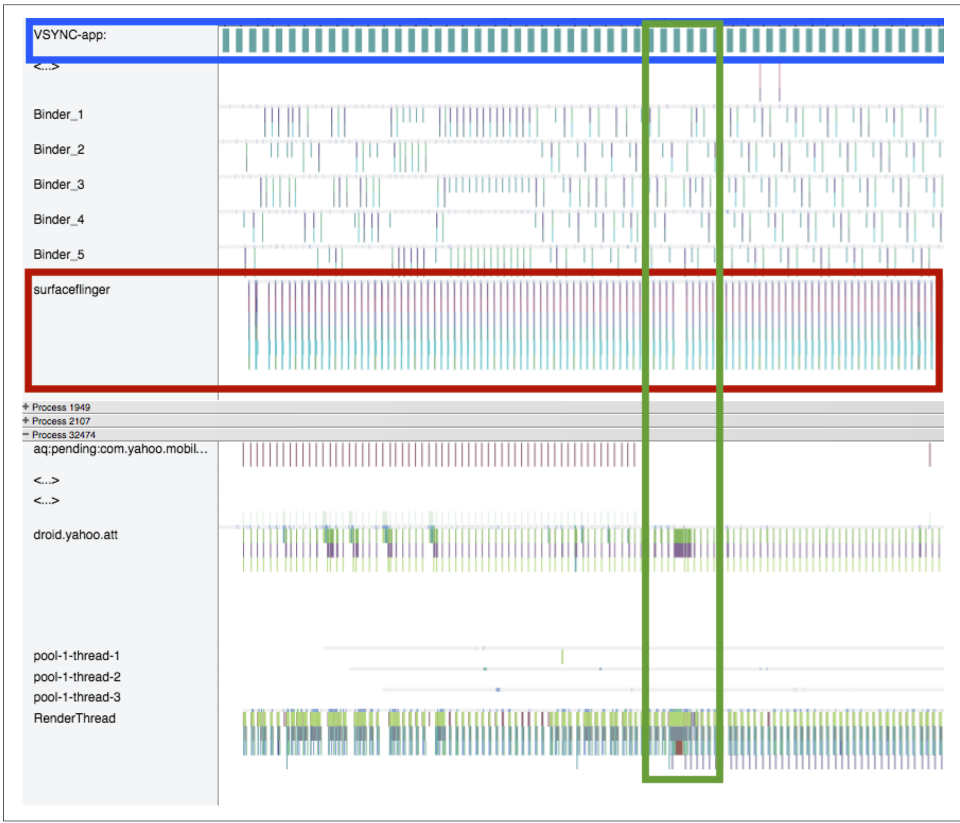


Figure 4-24. Digging into Systrace Jank (Lollipop)

The bottom section of Figure 4-24 shows details about the app. The second line of data (green and purple lines) are the app building views, and then the bottom row (green, blue, and some purple bars) is the RenderThread, where the views are rendered and sent to the buffer (not shown in this screenshot). Note that these bars get thicker at the same location as the potential jank in the surfaceflinger (about one-third of the way through the trace), indicating that something in the app may have been the cause of the jank. Each app is different, and will have a different cause, but these are the sort of symptoms we are looking for.

This high-level view is a great way to look for jank, but to investigate we must zoom in to get a better look. To understand what is happening in the Systrace, it is best to figure out what Systrace measures, and how things work when everything is working well. Once you figure out the way things *should* work, it makes finding the issues easier. In Figure 4-25, I have edited together the pertinent lines from a Systrace where things were running smoothly (taking out a lot of whitespace for space considerations). We start at the left side of the screen with droid.yahoo.com. Note that my

description will have you bouncing up and down in the trace to different lines (from the app to the OS) as the rendering occurs:

- Red box: droid.yahoo.com is finishing up a measure of the views for the screen. These are passed to the RenderThread.
- Orange box: RenderThread. Here the app:
  — Draws the frame (light green).
  — Flushes the Drawing buffer (gray).
  — Dequeue the buffer (in purple).
  — Sends this to a buffered list of views.
- Yellow box: com.yahoo.mobile.client.andr…

This is the list of views in a buffer. The height of the line demotes how many views are buffered. At the start, there is one, and when the view is passed to the buffer, the height doubles to two.

- Green box: VSYNC-sf alerts the surface flinger that it has 16 ms to render a screen. The brown bar on this line is 16 ms long.
- Blue box: surfaceflinger grabs a view from the queue (note in the yellow box that the buffer queue drops from 2 to 1). Upon completion, this view is sent off to the GPU and the screen is drawn.
- Purple box: VSYNC-app now tells the app to render another view (and shows a 16 ms timer).
- As soon as the VSYNC begins, the process repeats itself with droid.yahoo.att, measuring the views, passing to the RenderThread, and so on. And the cycle continues.

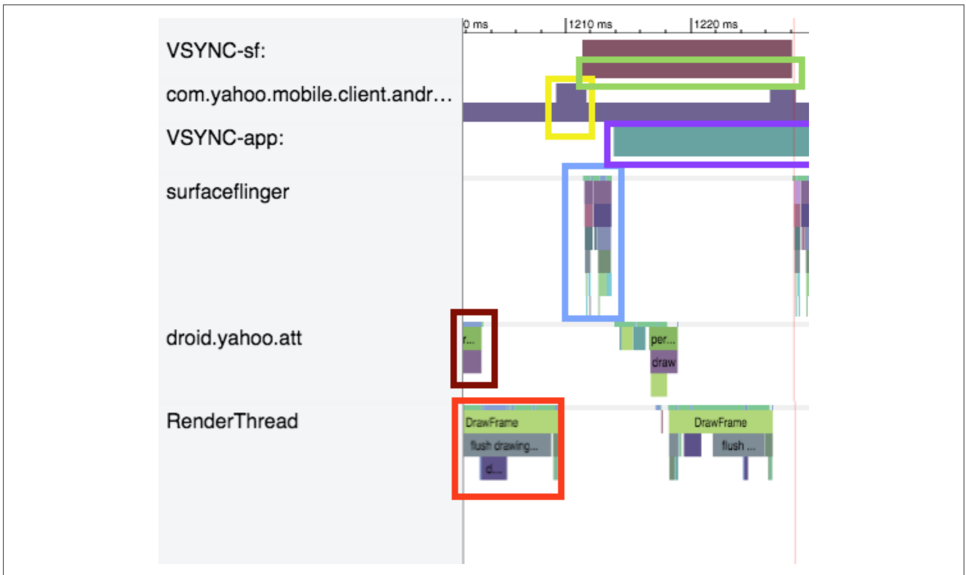


Figure 4-25. Systrace of proper rendering (Lollipop)

On reflection, it is pretty amazing all of the steps that our devices do to just render a screen so smoothly in such a short period of time. Now that we know what things look like when running smoothly, let's debug a moment of jank.

In Figure 4-26, we are looking at a close-up of the OS layer. To highlight the issue, I have added arrows indicating 16 ms intervals, and a red box at the location of a missing surfaceflinger.

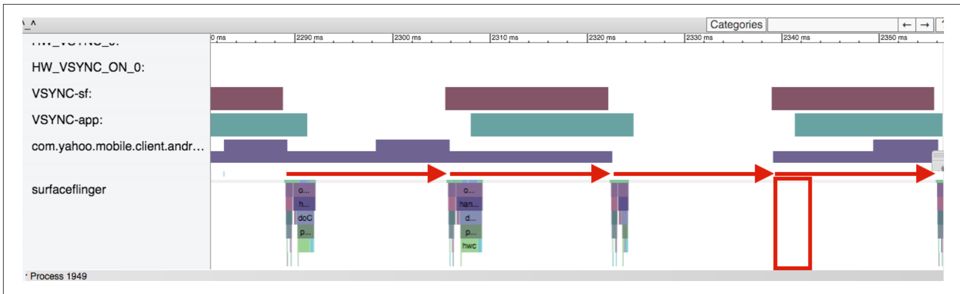


Figure 4-26. Systrace of jank—Operating System View (Lollipop)

Why does this happen? The row above the arrows is the view buffer, and the height of this row indicates how many screen frames are saved in the buffer. At the start of this trace, the buffer is alternating between one and two views. As the surfaceflinger grabs one view (the buffer count drops), but the buffer is quickly repopulated from the app.

However, after the third SurfaceFlinger action, the buffer queue empties and is not repopulated by the app in time. So, let's see what was happening at the app level.

In Figure 4-27, we initially see the RenderThread passing a view to the buffer (red box). The orange box shows the app creating a second view, rendering it, and passing it to the buffer (droid.yahoo.att measures and lays out the views, and RenderThread draws). Unfortunately, the app gets hung up before building another view (inside the yellow boxes). During the building of the next screen, the droid.yahoo.att app must first run (light green) "obtainView" for 7 ms, (teal) "setupListItem" for 8.7 ms, before the dark green "performTraversals" (3 ms). The app then passes the data to the RenderThread, which is also significantly slower (12 ms). Creating this frame took the app nearly 31 ms (versus ~6 ms for the previous view). When the process to build this frame began, there was one view stored in the buffer, but the device required two views during this time period. As the process had not fed the buffer, a jank occurred in the screen render.

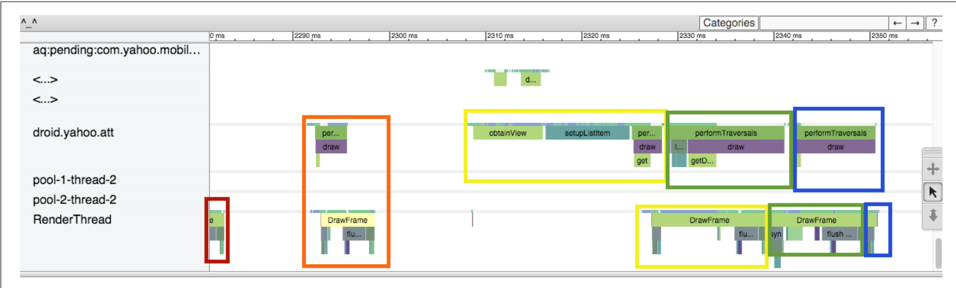


Figure 4-27. Systrace of jank—App View (Lollipop)

It is interesting to note that the app catches up quickly. After the delayed yellow box view is created and passed to the buffer, two additional frames are created in quick succession (green and blue boxes). By quickly refilling the buffer queue, the app survives with just one skipped frame. This trace was taken on a Nexus 6 (with a fast processor that allowed it to catch up quickly). Repeating this same study on a Samsung S4 Mini running Jelly Bean 4.2.2 resulted in the trace shown in Figure 4-28.

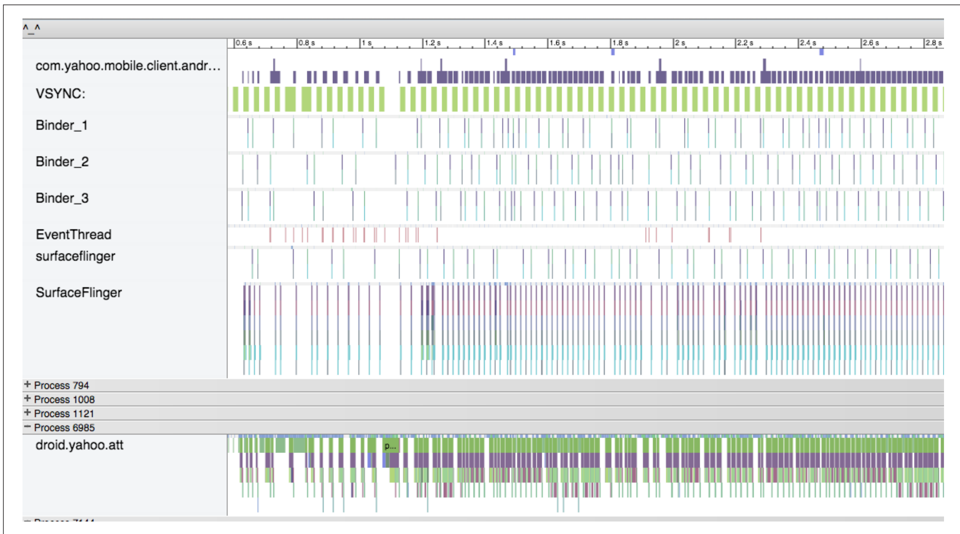


Figure 4-28. Systrace of jank (Jelly Bean)

It is immediately clear from the high-level view that many more frames are skipped (see the many gaps in the surfaceflinger at the start of the trace). Also good to notice is that the top row (the view buffer) often has zero views in its buffer (which we just saw leads to jank), and rarely has two views in the buffer. On a device with a slower GPU processor, the app does not have as many opportunities to "catch up" and refill the buffer like the Nexus 6 did.

You can exceed the 16.6 ms time to render a frame occasionally, as there are often one or two buffered frames ready to go. However, if you have two or three slow frame renders in a row, your customers will experience jank.

Because this trace was taken on a handset running Jelly Bean, the RenderThread data is included with the droid.yahoo.att row (the measure, draw, and layout are reported together until Lollipop). Combining these rows makes each step appear thicker. The small amount of whitespace between each call shows that this device has very little *extra* time between frame draws. The app on this device is only able to run slightly ahead of the surfaceflinger to keep the buffer queue full. If this app were able to reduce the complexity of each view—thus speeding up the rendering of the views—there would be more empty space between draws, the buffers would have more opportunity to fill, and likely would add a little "breathing room" in its view drawing on lower-end devices.

By highlighting a region, Systrace will count up all of the slices seen, and give basic statistical analyses by mousing over any of the values. In Figure 4-29, we see the performTraversals (the parent draw command) averages 13.8 ms, with a standard deviation of 5 ms. Because the 16 ms jank threshold lies within one standard deviation of the mean, we can guess that there is a jank problem on this device.

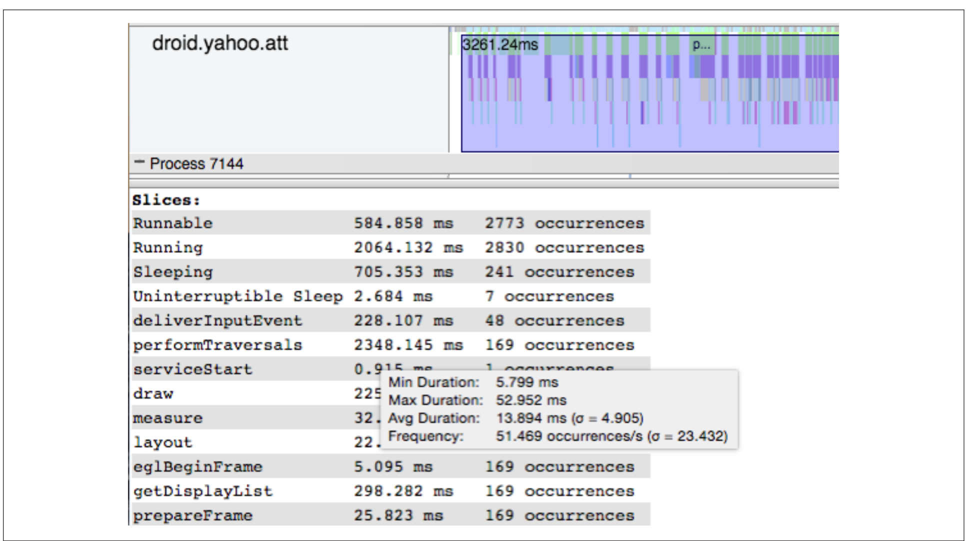


Figure 4-29. Systrace data summary

Zooming into this section shows this in detail (Figure 4-30). Each vertical red line indicates 16 ms. There are five or six instances here where the SurfaceFlinger misses the 16 ms mark. The length of the green "performtraversals" lines are all nearly 16 ms long (and because they must occur between each frame build, jank). There are also two blue-green deliverInputEvents (that take well over 16 ms each) block the app from drawing the screen.

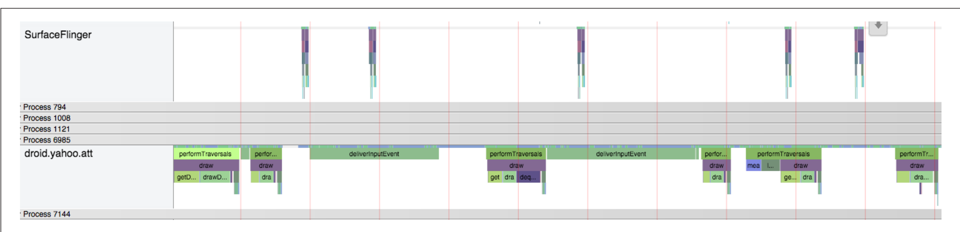


Figure 4-30. Systrace detail on a slower device (running Jelly Bean); note how the frame creation takes longer than 16 ms (red vertical lines), causing jank

So, what is causing those deliverInputEvents that are causing so much trouble? This is the user touching the screen, and forcing the ListView to build all of the views. This is blocking at the CPU level. Let's briefly cover what CPU blocking looks like (and cover it in more detail in Chapter 5).

Systrace and CPU Usage Blocking Render

If you see excessive jank, and are unable to see any significant differences in the rendering or surfaceflinger, you can investigate what processes are running on the CPU at the top of the Systrace. If you can isolate a certain feature or process that could be preventing your app from drawing, then you can look to remove that code from blocking the draw process (usually by removing it from the main thread). In the "Is it a goat?" app there is an option to enable a Fibonacci delay. When you turn this option on, the app calculates a very large Fibonacci number (recursively) for each row of goat data. As you might imagine, this is very slow and CPU intensive. Because the calculation is being done in a way that blocks the rendering of the views, it causes dropped frames when creating the view, and the scrolling is very janky. This is the example used in Figure 4-21 to show how the log reports skipped frames. Let's now dig deeper into Systrace to find the process calculating the Fibonacci numbers.

Let's start again with looking at a trace that runs properly. Figure 4-31 shows the "Is it a goat?" app on an N6 using the unoptimized layout.

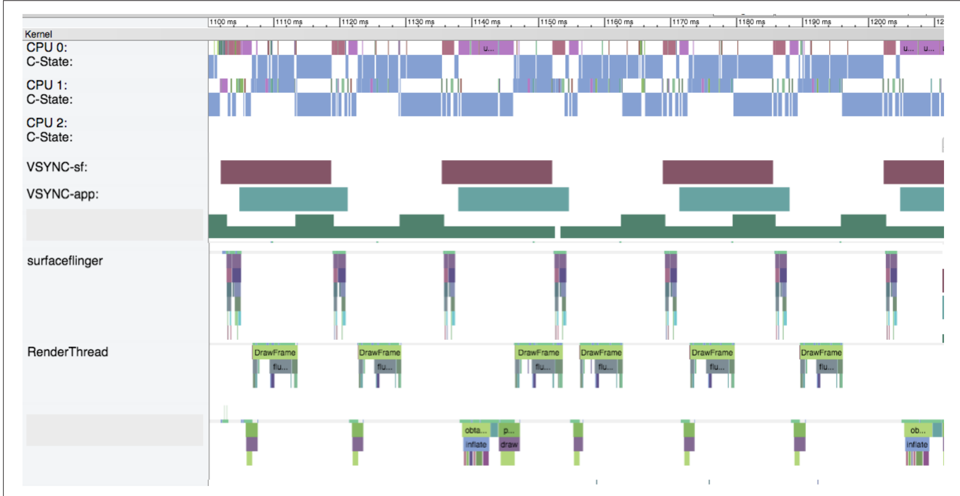


Figure 4-31. Systrace with CPU information

This view is modified, cutting out many lines between the CPU and surfaceflinger. In this trace, there is no jank, we see regular surfaceflingers every 16 ms (no jank). The RenderThread and Goat Process rows are creating all of the views and feeding them to the view buffer appropriately. Comparing these two rows to the CPU reveals a neat

pattern. When the RenderThread is drawing the layouts, CPU1 is running a blue activity (note that we are looking at the narrower CPU1, not CPU1:C-State). When the views are being measured by the Goat Process row, CPU0 has a corresponding purple process. They layouts are being built and drawn across two CPUs. Note that the major clicks on the X-axis are 10 ms each, and none of these processes take longer than 2–4 ms.

When we add the computationally intensive Fibonacci calculation into the draw, the Systrace takes a very different view (Figure 4-32).

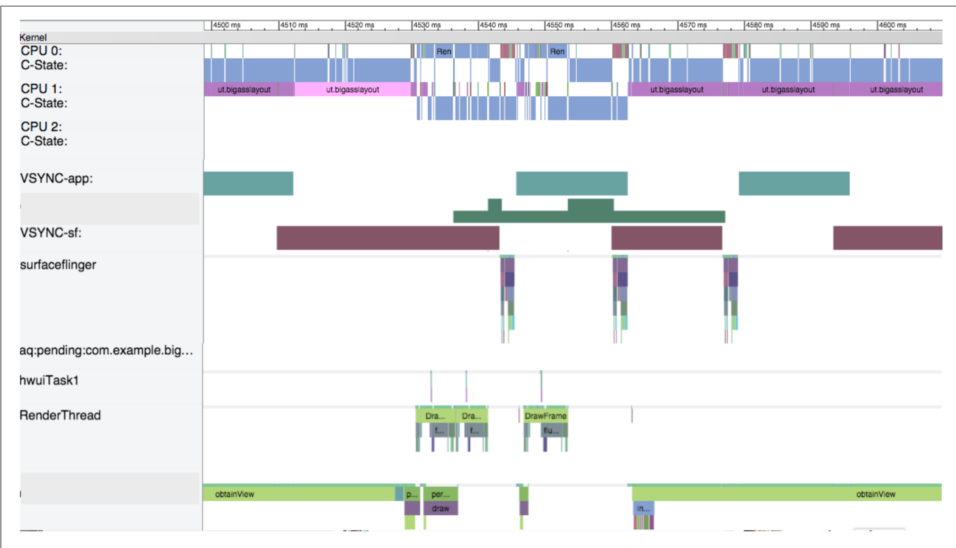


Figure 4-32. Systrace with CPU information and delay in rendering

This Systrace shows a lot of jank. In the same 100ms timeframe, only three surfaceflinger views are drawn (versus seven in non-delayed app). We can see the RenderThread is still drawing the views quickly (and you can see that in this trace, the blue RenderThread is running on CPU0). However, when measuring the views, the large recursive Fibonacci calculation is causing issues. The Goat Process row is spending most of its time in the obtainView state, rather than measuring. You can also see on CPU1 that the purple bands corresponding to Goat app processes are no longer 2–4 ms, but now range 2–17 ms long. The large Fibonacci calculations are taking 13–17 ms each, and this is really slowing down the app's ability to draw smoothly. We'll look at how to diagnose CPU performance (and its effect on rendering) in Chapter 5.

Systrace Update—I/O 2015

At Google I/O 2015, a new version of systrace was released that makes a lot of the analysis covered above a lot easier. In Figure 4-27, I highlighted each frame as it was updated. In the new version of systrace (shown in Figure 4-33), each frame is indicated by a dot with an "F" in it. Frames that render as expected have green dots, while slower (and very slow) frames are yellow or red. Selecting a dot and pressing m highlights the one frame for easier analysis.

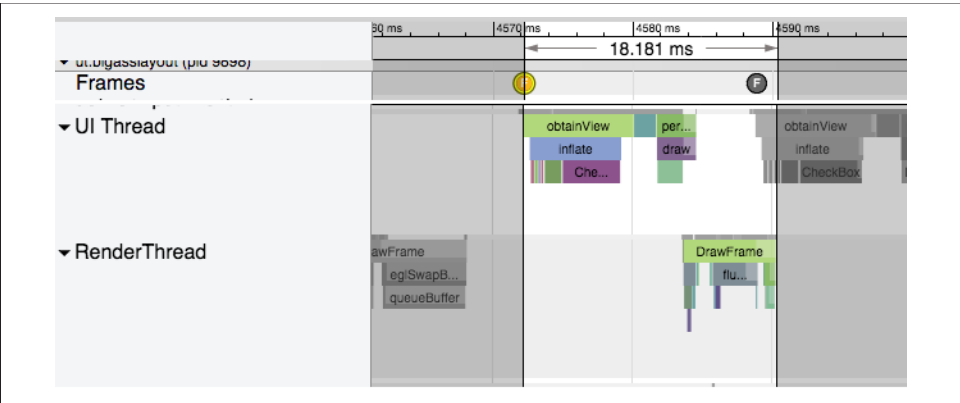


Figure 4-33. Systrace update with frame highlighted

The new Systrace also has a lot better descriptions as to what is happening. In Figure 4-33 the frame render time is 18.181 ms, and is colored yellow—as many frames in a row over 16 ms could lead to jank. In the description panel below the trace (shown in Figure 4-34), it warns that my app is recycling a ListView item, rather than creating a new one, and this is slowing down the view inflation.

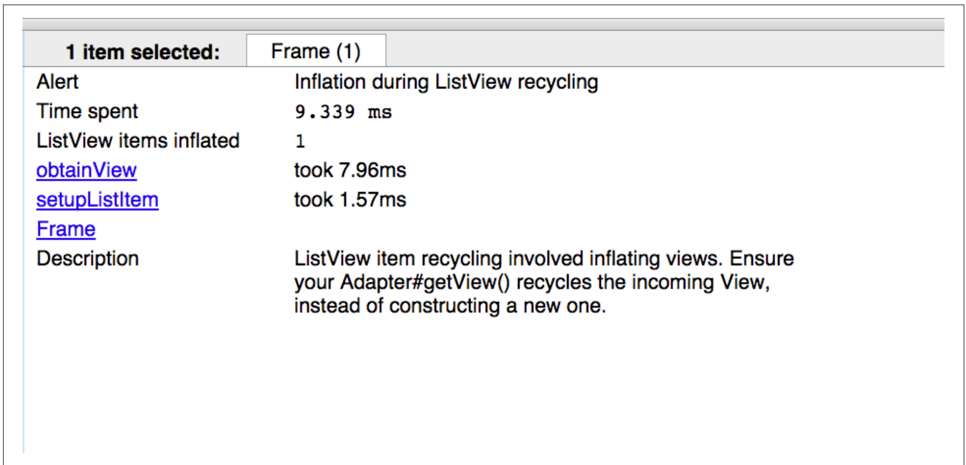


Figure 4-34. Systrace update with frame delay information

Alerts like these are shown as similar bubbles or dots inside Systrace, and also are listed in the alerts panel on the right side of the screen (shown in Figure 4-35).

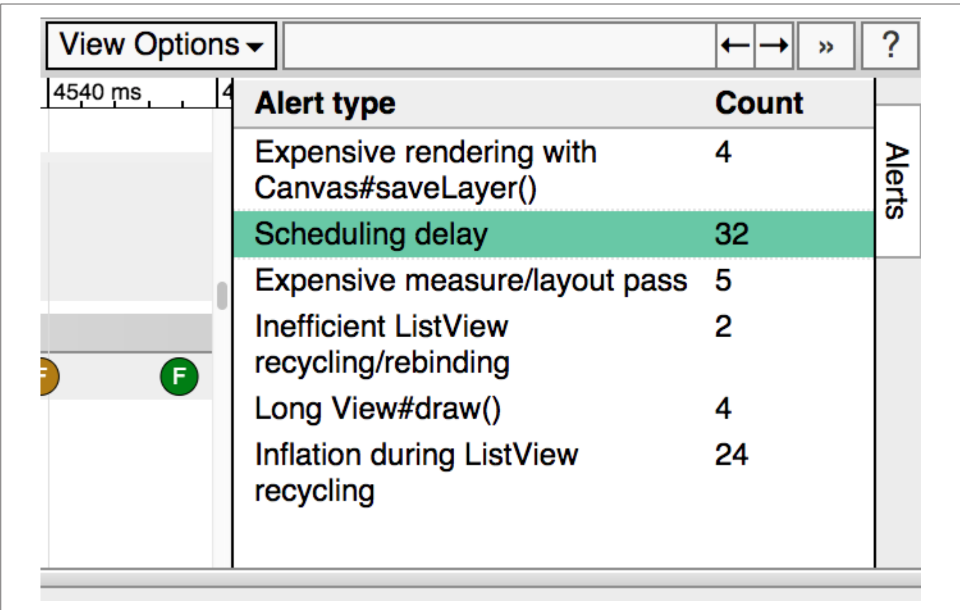


Figure 4-35. Systrace Update Alert panel

These new additions to Systrace make discovering issues slowing your UI even easier to diagnose.

Vendor-Specific Tools

Each of the major chip manufacturers have GPU profiling tools that can help you discover even more information about potential bottlenecks in rendering. These tools promise more detail into how your app runs on a specific chipset, allowing better tuning for these different GPU chips. This does go deeper than the scope of this book, but should the need arise, utilize these tools for even more powerful GPU debugging. Qualcomm, NVIDIA and Intel all offer special development tools to test your apps GPU performance on their processors.

Perceived Performance

In the previous sections, we discussed how to make your UI fast through testing, discovering issues, and optimizing layouts. But there is another possible way to make your Android UI faster: make it *appear* faster. Of course, it is crucial that you work to optimize all of the code, views, overdraw, and other issues that might affect your UI first to really make your app as fast as possible. However, once you have done that, there are still a few ways to make your app appear faster to your customers.

The human mind behaves in interesting ways, and by changing the perception of waiting, you can make the delay seem shorter to your users. This is exactly why grocery stores put trashy magazines in the checkout aisle, as having something to look at makes the delay seem shorter. If you can deliver content in a way to make the delivery appear seamless, more power to you. It may seem like a sleight of hand trick to make users *feel* like things are happening faster, but at the end it is the perception of how fast your app loads that matters. This is tricky to implement well, as some perceived performance optimizations have backfired, so always A/B test to ensure that these help your customers feel the speed of your app.

Spinners: The Good and the Bad

Spinners, progress bars, hourglass icons, and other tools to indicate a pause have been around for years. They have also been used to make apps and transitions feel faster. When loading an app with a progress bar, consider using a progress bar with an animation that moves in the opposite direction of loading complete. Research has shown that users are 12% more accommodating of the time with an animated scrollbar. Spinners that pulse faster generally make the wait time appear to be faster than a slowly moving spinner.

However, if you have a delay, adding a spinner is *not* always a good idea. The developers of the iOS app Polar noticed that there was a bit of delay in their app while rendering views on a page. Following conventional wisdom, they added a spinner to the page to show its users that something was happening while the page was rendering, but the responses were unexpected. Feedback and reviews began to arrive about how

the app was slower, and there was a lot of waiting for pages to load (note that the *only* change made was to add the spinner, the app was not actually any slower). The addition of a waiting indication allowed the customers to cue in that they were waiting. Removing that visual queue, the perception was that the app had sped up (again no code other than the spinner was changed). By changing the perception of the wait, the app became faster. Facebook found similar data: using a custom spinner in their iOS app made their load time appear longer than when they used a standard spinner.

Addition of a spinner should be accompanied by user testing to ensure that the results are expected. In general, spinners are acceptable when a delay is expected: opening a new page or downloading an image over the network. But if the delay will be short (say less than one second), you should consider omitting the spinner. In these cases, the addition of the spinner implies a delay that is not really there.

Animations to Mask Load Times

Clicking and seeing a blank screen gives your customers the perception of waiting. It is exactly for this reason that browsers keep the old page visible when you click a link. In mobile apps, you may not want to keep the old page visible, but a quick sweep animation might provide enough delay to get the next view ready for display. Observe this while using your favorite Android apps, and how many sweep in updated views from the bottom or from the side.

The White Lie of Instant Updates

If your customers make an update on the page, immediately change the data on the page, even if the data has not yet hit the server (of course, you need to ensure that these updates 100% do eventually get updated on the server). For example, when you "like" a photo on Instagram, the mobile view immediately updates the like status, even before a connection is established to the server and the backend is updated. They call this an "optimistic action," in that the update will appear on the website and be visible to friends within a few seconds (or minutes if in a tunnel, or area with low coverage), but the update will occur, and there is no need to wait for the server to update to update the UI. The mobile user does not feel obligated to wait to "make sure it worked."

An added advantage to instantly updating the UI without requiring the update to post on the server is that your app appears to function when coverage is intermittent (like when your train enters a tunnel on the commute home). Flipboard, in Offline Network Queue has presented its queueing architecture, which is used to upload changes made while offline, and this could easily be used to immediately change the UI, and update the backend a moment or two later.

Another performance trick (that is essentially the opposite of upload later) is to upload ahead of time. For apps like Instagram where large uploads of photos can add

delay updates to the main UI, you can begin uploading these large files early. Instagram realized that the slowest step in post creation was data entry. While the user adds text around the image post, Instagram uploads the photo to the server *before* the post is made public. Once the customer hits the post button, only the text and the post command needs to be uploaded, and the perception is that the upload happened in no time. To think of it another way, Instagram was able to answer the question "should we add a spinner?" by architecting its app to never need a spinner.

Tips to Improve Perceived Performance

When the speed of your app is improved by optimizing the code or views, you can measure the difference with a stopwatch. Some of the perceived performance gains (like Instagram's) can be measured with s stopwatch, but others (like the spinner examples) cannot. Because typical analytics or measurement tools cannot be used, these improvements will need to be put in front of users in order to identify if customers perceive the difference. Usability testing of some sort, whether with a wider team, A/B testing, or usability testing, will let you know if your changes please or further frustrate your users.

Conclusion

The user experience of your Android app is directly tied to the way it appears on the screen. If your app is slow to load or if the scrolling is not fast and smooth, your customers will be left with a negative perception of your app. In this chapter, we've covered examples of view hierarchy, and profiled how flattening and simplifying views speed rendering. We've considered overdrawing the screen, and the tools used to identify overdraw issues. For issues that require deeper digging (into CPU issues), Systrace is great at debugging and determining the issues causing jank. Finally, there are tricks that make your appear faster and more responsive through tricks in rendering, and moving CPU/network tasks out of the critical path of rendering. In the next chapter, we'll look at how optimizing and reducing the CPU usage of your app will improve the performance.

CHAPTER 5

Memory Performance

At the end of Chapter 4, we examined an issue where processes in the app blocked the UI thread, preventing the screen to update. In this chapter, we'll look at how to measure and better understand how your app uses memory. Memory leaks are a major cause of crashes on Android, and using the tools discussed in this chapter to diagnose issues will help you prevent such leaks. Let's kick off the discussion with memory management and tips to optimize, and then in the second half of the chapter, we'll cover how to minimize the CPU usage of your app.

Android Memory: How It Works

Before we can discuss how to improve the memory efficiency of your Android app, we need to start with the basics on how Android handles memory management. Once we get a solid background on that, we can understand some of the pitfalls and how to resolve them. To introduce some of the basic terms, let's get some simple information from your Android device.

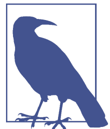
As you may be aware, the Java runtime on your Android device (whether Dalvik or ART) is a memory-managed environment. The runtime typically handles all memory allocations and cleanup (garbage collection). This does simplify the development of your app by abstracting those details from your code, but there are important considerations to take while building your app to ensure that the memory management works correctly.

Let's start with a quick set of definitions on the types of memory that are utilized by Android apps.

Shared Versus Private Memory

There are common framework classes, assets, and native libraries that are utilized by all apps. If every app had to individually keep these in memory, fewer apps could run concurrently. To save memory, Android uses shared memory for those assets. When attributing memory usage to an app, shared memory is averaged among all running processes.

Private memory is memory that is used just by your app and not used by other apps. Because this data is used by just your process, 100% of private memory is allocated to the process.

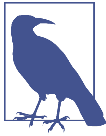


Zygote as Shared Memory

As you might recall from biology class, a zygote is the first cell created after fertilization—it splits into cells to become an embryo. Similarly, in Android, Zygote is a process that has all the framework classes, common assets, and native libraries preloaded inside of it. When your app starts, it is launched with a fork of the Zygote process (giving your app a head start with everything it needs from the system to survive) before loading any of your custom code. This allows your app to initialize faster than if it had to start from zero.

Dirty Versus Clean Memory

Dirty memory is memory that is only stored in the RAM, so if it were purged from the RAM, the app would need to be rerun to get the data back. Clean memory consists of items in RAM that are also saved on the disk, so if it were purged it could be easily reloaded from the device.



ART and Clean Memory

One of the main features of the ART runtime is that apps are compiled on install versus the Just in Time (JIT) of Dalvik. On devices running ART, now the app code is compiled at install and ready on disk. Recall that memory objects that can be accessed from disk are considered clean, and easy to remove when memory is low because it is easy to recover. Because now the app code in memory is now by definition clean, memory management is ART is further improved.

Currently, most devices are still using the Dalvik runtime, so the most common type of memory is private dirty memory (memory only used by one app, and only stored in memory).

Memory Cleanup (Garbage Collection)

Garbage collection (GC) is the act of cleaning up data objects that are no longer used so that the memory chunk can be reallocated to new objects. In general, once an object no longer has an active reference in the app, it can be garbage collected. The garbage collector begins with root objects (objects it knows are alive and in use by a process) and follows every reference looking for connections. If an object is not connected to the list of valid references, it must no longer be in use, and can be collected. Now the memory allocated to that object can be reused. In Figure 5-1, objects without active references (arrows) are colored in red, and will be removed when a garbage collection event occurs.

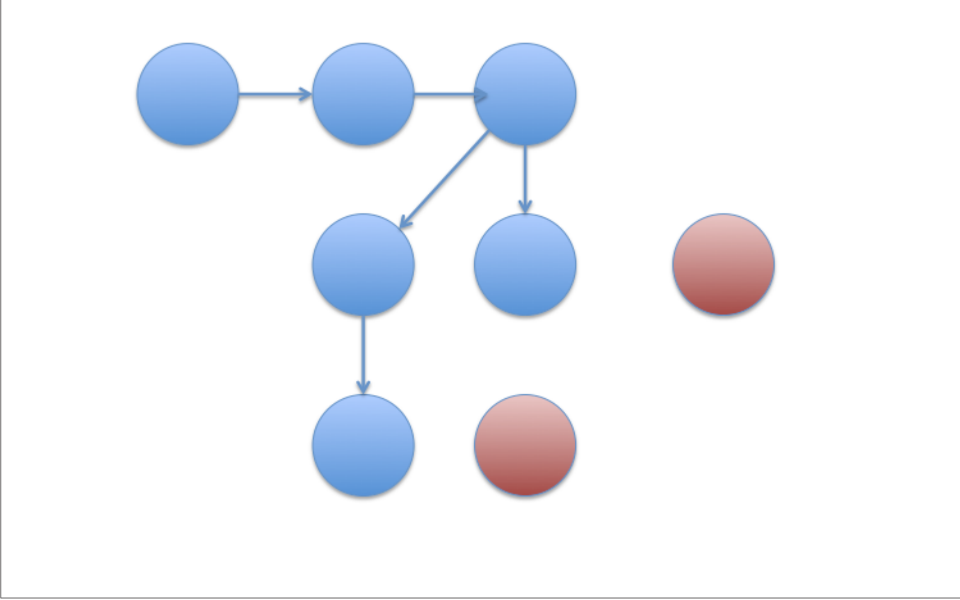


Figure 5-1. The Garbage Collector follows all references (arrows) marking active objects (blue), and collects all objects not currently referenced (red)

GC changes by OS

The GC in Android has evolved a great deal as Android has matured. Prior to Gingerbread, devices were low memory, so apps tended to have smaller heaps. The garbage collector was a "stop the world" collector meaning that GC caused all other processes and threads on the CPU to stop while the garbage was collected. This was a full heap collection, meaning that the collector traversed the entire heap looking for garbage. For low-memory apps, GCs were quick: maybe 2–5 ms, and may not have been noticeable. However, as devices grew more powerful (read more memory), and apps larger, garbage collection started to take longer. These pauses began interrupting the UI, meaning that the garbage collector needed to evolve.

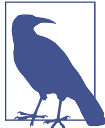
In Gingerbread, a concurrent GC that does just partial collections was instituted. Although a partial GC does not clean up all objects that are unreferenced, it is faster (because it does not travel the entire heap on each collection). Instead of stopping your app from running, the concurrent GC runs alongside your app. This means that now there are two short pause times at the start and end of each GC, but they are generally under 5 ms total. With shorter system stops, and no longer "stopping the world," your app is able to run alongside the GC—working to prevent GC from being a cause of jank in your app.

For devices running KitKat and earlier, garbage collection is simply "mark and sweep." The old objects are found and removed, but all other objects are left in place. This is shown in the first and second rows of Figure 5-2. When a GC is run, the allocated memory (shown in blue) removes the unreferenced objects, leaving small chunks of free RAM (the same size of the objects removed) in the allocated space. If the device has a small heap, or there are a lot of small collections, the device memory can get fragmented with small chunks of utilized and free RAM. Your device might tell you that you have 20 MB of free memory, but it does not tell you that the largest chunk of free RAM is actually only 1 MB. This will be a problem if you are trying to create a 4 MB bitmap—because you will get an out-of-memory error—there is not a 4 MB slice of RAM available for your object!

When the Android runtime changed from Dalvik to ART in Lollipop, the garbage collection was again improved. One point of the ART manifesto is: "Garbage collection should help, not hinder." GCs pause only once per collection (down from two, as instituted in Gingerbread), occur less often, and when they do run, they are significantly faster (online reports show that typical GCs have dropped from 10 ms to 3 ms). Further, in ART, large objects (like bitmaps) have their own special heap that is dedicated for large objects to simplify their memory management (and speeding GC of these big objects).

There are a number of new GC algorithms in ART, but one interesting method that is carried out when an app is no longer in the foreground is the Semi-Space GC. Because the app is not in the foreground, rewriting the objects in memory is safe—and as seen in the third line of Figure 5-2—after the unreferenced objects are removed, the used memory is copied to a free area of memory (without the small gaps). This allows for larger free chunks of memory to be opened up for other apps. When objects are moved in memory, the app has to be suspended to avoid errors. This can add to jank in an app, so the Semi-Space GC is only run with your app not in the foreground. This is not a fully compacting GC, but it is a very useful way to open up large areas of unused memory.

Figure 5-2. Garbage Collection: blue indicates memory in use, white indicates free space



The Future: Compacting GC

In 2015, there is a project in the AOSP to being a compacting garbage collector for a future release of Android. This will further reduce the number of memory problems as the memory locations can be moved around to defragment and free up large chunks of memory. A compacting GC takes the Semi-Space GC a step further: instead of simply forgoing a new memory space, it rewrites the objects in the same memory locations, but again without the small fragments. While this is an exciting future, you should make sure to *future proof* your C/C++ and NDK code by not referencing memory locations, as they may begin to slide around in the future.

The easiest way to find out if a GC has taken place is to look in your logs:

```
I/art     (10821): Explicit concurrent mark sweep GC freed
5124(199KB) AllocSpace objects, 1(16KB) LOS objects, 31% free,
34MB/50MB, paused 1.238ms total 23.656ms
```

This log report is showing a garbage collection run on process 10821 (which you'll see in subsequent pages is the "Is it a goat?" app). The GC was run concurrently with the process, pausing the UI once for 1.238 ms (so unlikely to cause any jank). The GC ran concurrently with the app for 23.6 ms. The app's heap is 50 MB, and is using 34 MB leaving 31% free. The GC freed 5,124 AllocSpace objects—relinquishing 199 KB —and cleaned up one Large Object from the Large Outpost Space of 16 KB (recall that large objects have their own dedicated memory in ART).

When does garbage collection occur?

Garbage collection occurs when the system feels it needs to reclaim memory. Perhaps your app has allocated new objects (increasing the memory requirements of your app), or perhaps new views are being created, and the old ones invalidated (releasing the references in memory). Perhaps your app has a memory leak, and keeps unused references in memory (preventing GCs, but causing other memory problems).

In the next section, we'll look at tools that will help you diagnose where your application is using memory, and locate memory leaks or code that generates excess garbage collection to ensure healthy memory usage on all Android devices.

Figuring Out How Much Memory Your App Uses

So we now have an idea of how memory is divided up inside an app, and how the system decides to clean up memory with garbage collection. While our apps are getting bigger and more complex, the guiding principle to memory management is to use as little as possible. The biggest consumers of memory (in general) are bitmaps. No matter how well you have compressed the file for network transmission, PNG and JPEG files use 32 bits per pixel, meaning that your 100 x 100 pixel thumbnail can use 320,000 bits of memory. Loading a number of these images at once, and you see how apps are using 50–100 MB of your memory heap.

How much memory can you use on a device? `ActivityManager.getMemoryClass` will return the maximum size for your app's heap. If this is smaller than what you have found to be ideal, you can reduce the content displayed, or perhaps scale the images to a smaller format. You can request the `getLargeMemoryClass()` if you will be building a memory-intensive app, but it should be used with care as a large memory heap will actually slow down your app during garbage collection events (as the framework will have to hunt through more data for unused objects). How do we find out how our app is using memory?

Running `adb shell dumpsys meminfo` on my Nexus 6 (with the "Is it a goat?" app in the foreground) gives the following information:

```
Applications Memory Usage (kB):
Uptime: 7009870 Realtime: 7218457

Total PSS by process:
   522515 kB: com.amazon.mShop.android (pid 5610 / activities)
   520153 kB: com.coffeestainstudios.goatsimulator (pid 19139 / activities)
   207397 kB: com.facebook.katana (pid 9430 / activities)
   183514 kB: com.android.systemui (pid 2111 / activities)
   141205 kB: com.example.isitagoat (pid 10821 / activities)
   113143 kB: com.google.android.googlequicksearchbox (pid 2471 / activities)
    99168 kB: system (pid 1957)
    61157 kB: com.rovio.gold_ama (pid 18842 / activities)
    58917 kB: com.amazon.kindle (pid 19331)
    49859 kB: surfaceflinger (pid 248)
    48874 kB: com.elvison.batterywidget (pid 2983)
    48270 kB: com.urbandroid.lux (pid 5656 / activities)
    35940 kB: com.facebook.orca (pid 4441)
    32541 kB: com.google.android.apps.plus (pid 20233)
    26461 kB: com.google.process.gapps (pid 2545)
    25989 kB: com.google.android.googlequicksearchbox:search (pid 2586)
    23893 kB: com.google.android.gms (pid 2610)
```

Proportional Set Size (PSS) memory is the total memory used by your app. Recall that the total memory is all of the private memory (shown in some reports as "USS - Unique Set Size") plus a percentage of the shared memory. In this case, there are several apps in the background that still use more memory than the 141,205 KB of the "Is it a goat?" app. Note that the PID for the "Is it a goat?" app is 10,821, as this identifier is used by Android to identify this app.

The next section of the report breaks down the memory usage even further. First, we see how much memory is being used by the system for rendering views (remember surfaceflinger from "Systrace" on page 103?). The media server also uses a lot of memory, but there are hundreds of small processes in the native memory (the list was truncated for space concerns). After Native and System are apps that appear as "persistent" processes that are always running on the device—the system UI, NFC, and phone.

The next sections are where we can see the apps actually running on the device. In the foreground you can see the "Is it a goat?" app. Visible and perceptible apps are apps that have some presence on the screen (either as a notification in the case of the battery widget), or as an overlay (in the case of the lux app).

A Services, B Services, and Cached apps are all apps that are in the background, but have memory allocated to their processes. Either they have run in the past, and will be cleaned up during a time of memory pressure, or they do occasionally run in the background:

```
Total PSS by OOM adjustment:
   105030 kB: Native
               49859 kB: surfaceflinger (pid 248)
               17010 kB: mediaserver (pid 1539)
                4785 kB: rild (pid 1537)
                3555 kB: logd (pid 243)
                3494 kB: mm-qcamera-daemon (pid 1553)
                3466 kB: zygote (pid 1546)
                2405 kB: gsiff_daemon (pid 1549)
                1669 kB: sensors.qcom (pid 250)
                1610 kB: drmserver (pid 1538)
                1407 kB: thermal-engine (pid 1545)
                1260 kB: ks (pid 768)
                1188 kB: netd (pid 1535)
                1128 kB: sdcard (pid 1550)
                1072 kB: wpa_supplicant (pid 2188)
                                //plus a lot more

    99168 kB: System
               99168 kB: system (pid 1957)
   221364 kB: Persistent
              183514 kB: com.android.systemui (pid 2111 / activities)
               16764 kB: com.android.nfc (pid 2418)
               16231 kB: com.android.phone (pid 2442)
```

```
              4855 kB: com.android.server.telecom (pid 2392)
   141205 kB: Foreground
            141205 kB: com.example.isitagoat  (pid 10821 / activities)
    60554 kB: Visible
             26461 kB: com.google.process.gapps (pid 2545)
             19900 kB: com.google.process.location (pid 2917)
              9669 kB: com.google.android.inputmethod.latin (pid 2304)
              4524 kB: com...googlequicksearchbox:interactor (pid 2270)
    97144 kB: Perceptible
             48874 kB: com.elvison.batterywidget (pid 2983)
             48270 kB: com.urbandroid.lux (pid 5656 / activities)
    16113 kB: A Services
              8538 kB: com.google.android.gms.wearable (pid 3056)
              7575 kB: android.process.media (pid 29108)
   113143 kB: Home
            113143 kB: com...googlequicksearchbox (pid 2471 / activities)
   859266 kB: B Services
            522515 kB: com.amazon.mShop.android (pid 5610 / activities)
            207397 kB: com.facebook.katana (pid 9430 / activities)
             58917 kB: com.amazon.kindle (pid 19331)
             35940 kB: com.facebook.orca (pid 4441)
             25989 kB: com...googlequicksearchbox:search (pid 2586)
              4317 kB: org.simalliance.openmobileapi.service:remote (pid 4903)
              4191 kB: com.android.sdm.plugins.sprintdm (pid 14923)
   809634 kB: Cached
            520153 kB: com...goatsimulator (pid 19139 / activities)
             61157 kB: com.rovio.gold_ama (pid 18842 / activities)
             32541 kB: com.google.android.apps.plus (pid 20233)
             23893 kB: com.google.android.gms (pid 2610)
             22116 kB: com.levelup.touiteur (pid 19038)
             18309 kB: com.mobileiron (pid 26851)
             17872 kB: com.linkedin.android (pid 27259)
             15763 kB: com.amazon.mShop.android.shopping (pid 24968)
             15177 kB: com.google.android.apps.magazines (pid 26772)
             14078 kB: android.process.acore (pid 26874)
             13740 kB: com.google.android.music:main (pid 24911)
             13280 kB: com.android.mi.email (pid 26748)
             12072 kB: com.yahoo.mobile.client.android.mail.att:
com.yahoo.snp.service (pid 26904)
             10377 kB: com.alphonso.pulse (pid 26441)
              5504 kB: com.android.chrome (pid 24943)
              4823 kB: com.google.android.deskclock (pid 28469)
              4717 kB: com.android.cellbroadcastreceiver (pid 20193)
              4062 kB: com.android.defcontainer (pid 28116)
```

Finally, the report breaks down the total memory usage by type of memory, and the breakdown of free versus used RAM. In the case of my Nexus 6, it is clear that the apps cached in memory will likely stay in memory, as nearly 50% of the RAM is still free:

```
Total PSS by category:
   789741 kB: Unknown
   501966 kB: Dalvik
   463460 kB: GL
   225937 kB: Other dev
   204576 kB: Graphics
    74916 kB: Ashmem
    63123 kB: .so mmap
    56944 kB: .dex mmap
    41319 kB: image mmap
    36328 kB: Dalvik Other
    22039 kB: code mmap
    20906 kB: .apk mmap
    12460 kB: Stack
     4998 kB: Other mmap
     3716 kB: .jar mmap
      112 kB: Cursor
       56 kB: .ttf mmap
       24 kB: Native
        0 kB: Memtrack

Total RAM: 3041412 kB (status normal)
 Free RAM: 1465830 kB (809634 cached pss + 450524 cached + 205672 free)
 Used RAM: 1967459 kB (1712987 used pss + 71340 buffers +
                       101780 shmem + 81352 slab)
 Lost RAM: -391877 kB
   Tuning: 256 (large 512), oom 325000 kB, restore limit 108333 kB (high-end-gfx)
```

This is a great overview to the memory usage on your device, but you are probably more interested in the details for just *your* app. To learn more about the amount of RAM your app is currently using, you can add the PID number to the `meminfo` command:

```
adb shell dumpsys meminfo 10821
Applications Memory Usage (kB):
Uptime: 10475753 Realtime: 10684340

** MEMINFO in pid 10821 [com.example.isitagoat] **
                   Pss  Private  Private  Swapped     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap        0        0        0        0    13752    13752    29255
  Dalvik Heap    13639    13080        0        0    42782    34636     8146
 Dalvik Other      556      556        0        0
        Stack      132      132        0        0
    Other dev     6622     6592        4        0
     .so mmap     1082      164       60        0
    .apk mmap       52        0        0        0
    .ttf mmap        0        0        0        0
    .dex mmap        8        0        8        0
    code mmap      471        0       16        0
   image mmap      832      532        0        0
```

```
  Other mmap        17        4        0        0
    Graphics     66784    66784        0        0
          GL     26356    26356        0        0
     Unknown     11799    11736        0        0
       TOTAL    128350   125936       88        0    56534    48388    37401

Objects
              Views:      121         ViewRootImpl:        1
        AppContexts:        3           Activities:        1
             Assets:        2        AssetManagers:        2
      Local Binders:        8        Proxy Binders:       16
   Death Recipients:        0
    OpenSSL Sockets:        0

SQL
        MEMORY_USED:        0
 PAGECACHE_OVERFLOW:        0          MALLOC_SIZE:        0
```

This is the memory usage of the "Is it a goat?" app while it is in the foreground on a Nexus 6. Let's break down what the previous table is telling us. We'll only worry about the data in the first two columns: the total memory in use by the app (recall PSS = shared + private memory) and the private dirty memory (memory only in use by the app, and not stored on disk). Note that most all of the memory allocated is private dirty data:

- 13 MB from Dalvik (which I assume should read ART, but was not changed)
- 66.7 MB is allocated to graphics
- 26 MB are dedicated to GL commands of rendering
- 11.8 MB is unknown
- 66 MB are dedicated to "other dev"
- Smaller allocations (most of which are smaller shared resources)

Total memory usage is 128 MB. In ART, graphics are stored in a new "large object space" in the main heap. This larger space allows for better garbage collection, and less fragmentation for bitmaps, which are generally the largest objects in your apps memory, allowing your heap to be smaller.

In the second table, there is information about things that are using memory: the view count, asset count, and the number of activities. If for some reason, these numbers are much higher than you expect, wait for a second and run `meminfo` again. A garbage collection may clean up views that were recently invalidated. If they remain (or if the count grows as you use the app), you likely have a memory issue to investigate. You'll also be able to see memory allocated for databases, and other files used by your app here.

Procstats

The `meminfo` command gives a lot of amazing information for one instant in time. Memory leaks generally occur over time, and correlating multiple `meminfo` reports would be cumbersome. In KitKat, procstats was introduced to help you understand how much memory your app uses in the background over a set period of time. In Settings → Developer Options → Process Stats, you can see a visual readout of your device's memory usage (the default timeframe is for the last 3 hours, but you can change it to 6, 12, or 24 hours). In Figure 5-3, the top of the screen tells you the current state of the device's memory, and the bar is an indicator of memory usage over time (green, yellow, and red bars indicating severity of the memory issues). Clicking this bar provides more details: the time spent in each memory state, and how the memory is being allocated. If you'd like to see memory usage for foreground or cached apps, you can change the stats type from the settings menu.

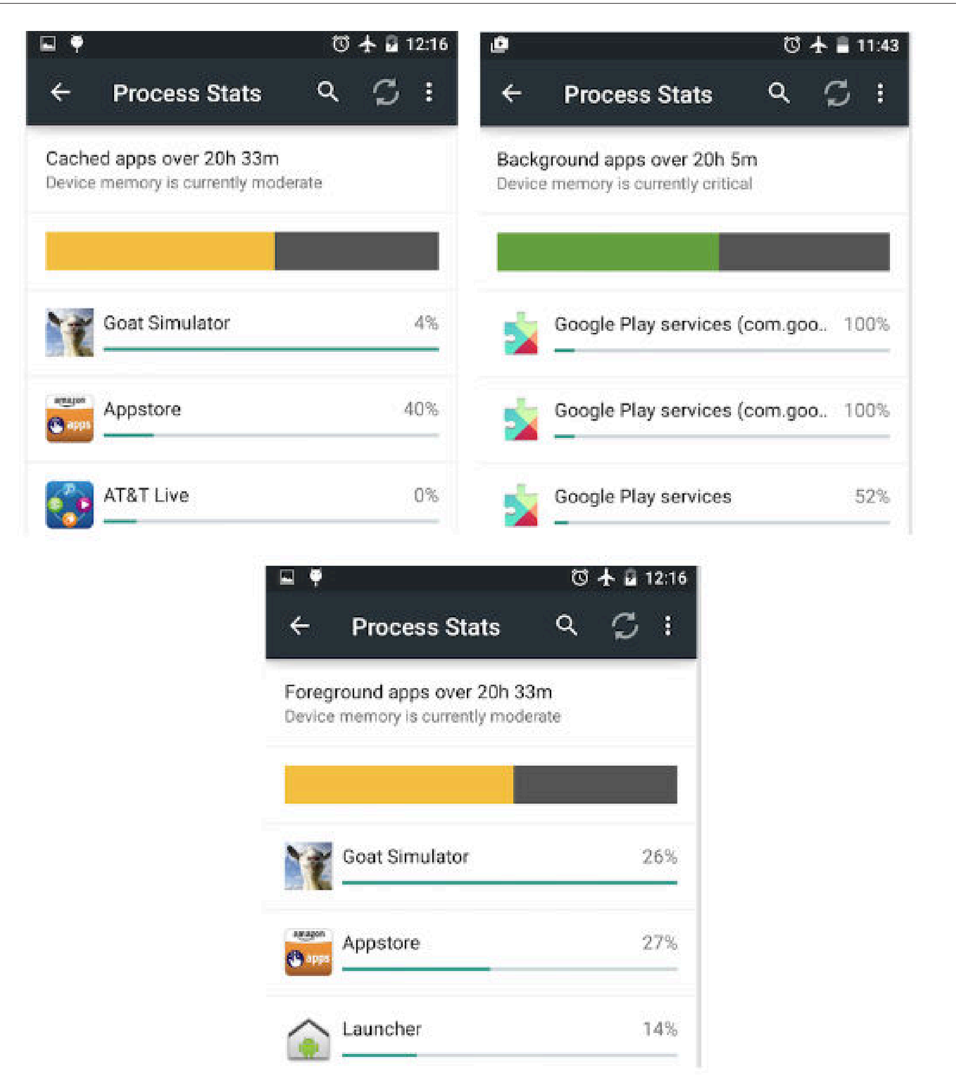


Figure 5-3. Procstats overview of app memory usage while cached (top left), in the background (top right), and in the foreground (bottom)

Each running app is listed with the percentage of time it has been active, and the bar is a comparison of the average memory used by each app (again, you can see this for foreground, background, or cached). Clicking on an app gives you detailed information about how your app uses memory, and the RAM and runtime in the state of the parent menu. To see how your app performed in another state (e.g., foreground to cached, like in Figure 5-4), you must go back to the main menu to change the state, and then reselect your app. Due to the difficulty to navigate these menus for one app,

I typically use the command line version `adb dumpsys procstats` to get the table of data.

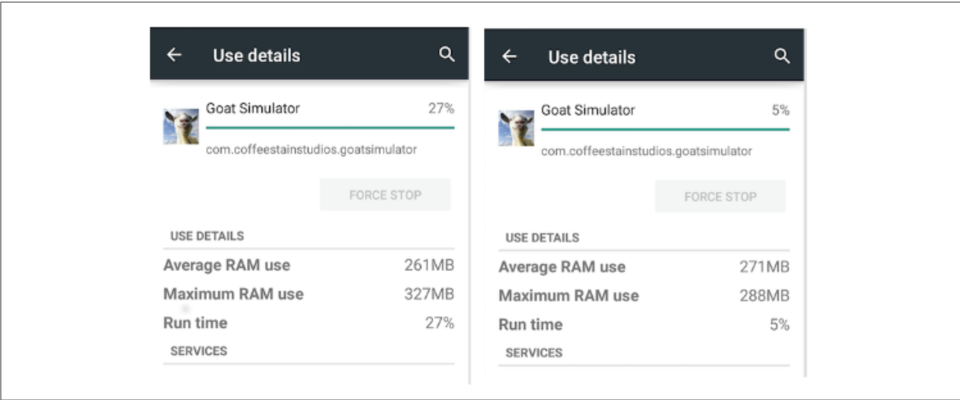


Figure 5-4. Procstats for app: foreground (left) and cached (right), shown in Lollipop

Compare Figure 5-4 to the wealth of information generated at the command line. The dump contains the stats for the last 24 hours, 3 hours, and current stats in all of the states. For space considerations, I'm only showing the data for the last 3 hours (the longer periods look similar). The first set of data is a breakdown of system usage with the screen off (SOff) or on (SOn) (Example 5-1).

Example 5-1. Procstats System Info

```
$ adb shell dumpsys procstats com.coffeestainstudios.goatsimulator

AGGREGATED OVER LAST 3 HOURS:
System memory usage:
  SOff/Norm: 1 samples:
                      //similar to SOn
       Mod : 1 samples:
                      //similar to SOn
       Crit: 1 samples:
                      //similar to SOn
  SOn /Norm: 3 samples:
        Cached: 304MB min, 317MB avg, 336MB max
        Free: 32MB min, 44MB avg, 57MB max
        ZRam: 0.00 min, 0.00 avg, 0.00 max
        Kernel: 41MB min, 46MB avg, 50MB max
        Native: 45MB min, 49MB avg, 50MB max
       Mod : 1 samples:
        Cached: 182MB min, 182MB avg, 182MB max
        Free: 24MB min, 24MB avg, 24MB max
        ZRam: 0.00 min, 0.00 avg, 0.00 max
        Kernel: 41MB min, 41MB avg, 41MB max
        Native: 46MB min, 46MB avg, 46MB max
```

```
      Low : 3 samples:
       Cached: 186MB min, 226MB avg, 287MB max
       Free: 19MB min, 104MB avg, 269MB max
       ZRam: 0.00 min, 0.00 avg, 0.00 max
       Kernel: 38MB min, 38MB avg, 39MB max
       Native: 46MB min, 47MB avg, 47MB max
      Crit: 5 samples:
       Cached: 146MB min, 179MB avg, 247MB max
       Free: 16MB min, 57MB avg, 130MB max
       ZRam: 0.00 min, 0.00 avg, 0.00 max
       Kernel: 38MB min, 40MB avg, 45MB max
       Native: 43MB min, 46MB avg, 49MB max
<big snip>

Summary:
<snip>
Run time Stats:
  SOff/Norm: +1h12m4s41ms
       Mod : +3m0s428ms
       Low : +1s954ms
       Crit: +1m7s324ms
  SOn /Norm: +27m26s70ms
       Mod : +6m9s749ms
       Low : +7m58s126ms
       Crit: +23m52s476ms
      TOTAL: +2h21m40s168ms
```

As the memory of the device moved from normal to moderate low and critical, the cached memory was purged to allow the active process to continue running (the average cached memory drops from 304 MB to 146 MB from normal to critical with the screen on). At the bottom of the dump is a Summary, which breaks down the 3 hour bucket of time in to the various memory states. It shows that the device was running for 2 hours 21 minutes of the 3 hour sample. While the screen was off, the device was primarily in a normal memory state, and when the screen was on, the device was in a low or critical memory state over 50% of the time.

What caused the device to enter these low memory states? By looking at the Goat Simulator specific report (below), we can start to piece together where the memory issues occur. The first table shows the process, and then a series of MB data. It shows that the app was the foreground (TOP) app for 15% of the time and (2.5% as the last active process). The memory numbers are reported in MB, and have the format `(total memory Low-Average High/Private memory Low/Average/High)`:

```
    Per-Package Stats:
      * com.coffeestainstudios.goatsimulator / u0a82 / v915134:
          * com.coffeestainstudios.goatsimulator / u0a82 / v915134:
                   TOTAL: 15% (119MB-261MB-327MB/113MB-255MB-321MB over 23)
                     Top: 15% (119MB-261MB-327MB/113MB-255MB-321MB over 23)
              (Last Act): 2.5% (260MB-273MB-292MB/256MB-268MB-287MB over 3)
                (Cached): 2.7% (268MB-271MB-288MB/263MB-267MB-284MB over 7)
```

The next section looks similar to the total system memory charts, but broken down to just the Goat Simulator process, first showing the time the process spent in different memory states with the screen off and on. The "Run time Stats" summary in Example 5-1 tells us that with the screen off, the device was in a critical memory state for 1 minute 7 seconds. Looking at the Goat Simulator, it was active, or the last active app for 46 s + 21 s = 67 s. The same holds for screen on: device critical for nearly 24 min, and Goat Simulator top or last accessed in a critical state for 23 minutes. This indicates that the memory state of the device might be related to the memory usage of this app.

In Example 5-2, below the timing, we get an additional memory usage breakdown of the app in the various states:

Example 5-2. Procstats App Info

```
Multi-Package Common Processes:
  * com.coffeestainstudios.goatsimulator / u0a82 (16 entries):
        SOff/Norm/LastAct: +1m32s937ms
             Mod /LastAct: +76ms
             Low /LastAct: +1s870ms
             Crit/Top    : +45s940ms
                  LastAct: +21s384ms
        SOn /Norm/Top    : +20s540ms
                  LastAct: +9s755ms
             Mod /Top    : +8s70ms
                  LastAct: +11s263ms
                  CchAct : +1s571ms
             Low /Top    : +21s335ms
                  LastAct: +10s802ms
                  CchAct : +3m31s584ms
             Crit/Top    : +20m0s324ms
                  LastAct: +2m55s742ms
                  CchAct : +18s8ms
                  TOTAL  : +30m51s201ms
        PSS/USS (10 entries):
          SOff/Crit/Top    : 1 samples 275MB 275MB 275MB / 270MB 270MB 270MB
                    LastAct: 1 samples 266MB 266MB 266MB / 261MB 261MB 261MB
          SOn /Norm/Top    : 1 samples 136MB 136MB 136MB / 127MB 127MB 127MB
               Mod /Top    : 1 samples 174MB 174MB 174MB / 167MB 167MB 167MB
                    LastAct: 1 samples 251MB 251MB 251MB / 248MB 248MB 248MB
               Low /Top    : 2 samples 155MB 201MB 247MB / 150MB 196MB 242MB
                    LastAct: 1 samples 260MB 260MB 260MB / 256MB 256MB 256MB
                    CchAct : 7 samples 268MB 271MB 288MB / 263MB 267MB 284MB
               Crit/Top    : 18 samples 119MB 279MB 327MB / 113MB 273MB 321MB
                    LastAct: 1 samples 292MB 292MB 292MB / 287MB 287MB 287MB
```

```
Summary:
  * com.coffeestainstudios.goatsimulator / u0a82 / v915134:
           TOTAL: 15% (119MB-261MB-327MB/113MB-255MB-321MB over 23)
             Top: 15% (119MB-261MB-327MB/113MB-255MB-321MB over 23)
      (Last Act): 3.8% (251MB-267MB-292MB/248MB-263MB-287MB over 4)
        (Cached): 2.7% (268MB-271MB-288MB/263MB-267MB-284MB over 7)

<snip>

          Start time: 2015-01-23 15:38:18
  Total elapsed time: +21h33m58s23ms (partial) libart.so

          Start time: 2015-01-24 11:48:20
  Total elapsed time: +1h23m56s121ms (partial) libart.so
```

When you inject an object into memory, the Android system will allocate memory for your object, and when the object is no longer in use, will reclaim the memory with garbage collection event. We discussed how memory cleanup works in "Memory Cleanup (Garbage Collection)" on page 123, and how to determine if your app is using excessive amounts to memory. The `procstats` command provides information about the state of the device's memory state. In the next section, we'll look at how you can use these warnings in your app to ensure your app continues to run when free memory is at a premium.

Android Memory Warnings

The Android system allocates the memory heaps available to each app, and also is tasked with keeping the memory garbage collection (removing old content from memory). In the previous section, we saw procstats reports showing the memory was reaching critical levels. When your app is running (or in the cache), it can listen to these reports, and free memory to prevent the process from being cleaned up for memory usage. `onTrimMemory` will tell you where your app is in the cache, and how you can help remove memory to prevent your entire app from being removed. If your app is running, and there are memory problems, `onTrimMemory` will issue the following warnings:

`TRIM_MEMORY_RUNNING_MODERATE`
: This is your first warning.

`TRIM_MEMORY_RUNNING_LOW`
: This is like the yellow light. It is your second warning to begin to trim resources to improve performance.

`TRIM_MEMORY_RUNNING_CRITICAL`
: This is the red light. If you keep on executing without clearing up memory resource, the system is going to begin killing background processes to get more memory for you. Unfortunately, that will lower your app's performance.

`TRIM_MEMORY_UI_HIDDEN`
: Your app was just moved off the screen, so this is a good time to release large UI resources. Now your app is on the list of cached apps. If there are memory problems, your process may be killed. Being a background app, release as much as you can so that your app can resume faster than a pure restart. There are three levels:

`TRIM_MEMORY_BACKGROUND`
: Your app is on the list, but near the end.

`TRIM_MEMORY_MODERATE`
: Your app is in the middle of the kill list.

`TRIM_MEMORY_COMPLETE`
: This is the "your app is next to be killed" warning.

In Example 5-2 (the two lines above the "Summary") for Goat Simulator, you can see that when memory is critical, and the app goes from "Screen on" top, foreground to the background, and "last active," the max memory usage drops from 327 MB to 292 MB.

Memory Management/Leaks in Java

When it comes to memory management, the first rule is to always minimize the amount of information you store in memory. By reducing your memory footprint, you are less likely to experience any memory-related error, and with fewer objects in memory, fewer objects are recycled, leading to faster garbage collection. We'll see some examples later in the chapter on how additional objects affect memory usage and app performance.

Even though memory in the Android runtime is managed, developers must still worry about how memory is being used. Memory leaks are possible in Android apps when objects are unnecessarily left in memory. This can happen due to accidental references or other links between activities that keep the GC from collecting the object. These accidental references can lead to out-of-memory issues on lower-memory devices, so it is critical that they are tracked down and resolved. If you see memory issues in your app (or you see unexpected results in the tools we have already discussed), you may have a leak, and you may need to dig further to discover and eliminate the leak.

Tools for Tracking Memory Leaks

The aforementioned `meminfo` tool can be useful in ascertaining if you have a memory leak. If the results from `meminfo` or Process Stats are surprising (more memory use in the background than you'd expect, or memory usage is increasing unexpectedly), there are additional tools to help you discover where your memory is leaking. Each leak will be unique, and the path to discover them will be different for each codebase, but these examples should help you start in the right direction.

Heap Dump

So how much data does your app actually use in memory? What sort of files are allocated into memory when your app runs? A great tool to better understand this information is the Heap Dump in Monitor DDMS. To activate the Heap Dump, select your app, and enable the Update Heap button—it's a cylinder half filled with green liquid (Android blood?). This will populate the menus and buttons on the right side of the screen. To discover how much memory your app is using, click the Cause GC button. This forces your app to run a garbage collection, cleaning up some files. The files that remain are counted by type and size, and reported into the Heap tool (run on a Nexus 7 on Android 5.0.2); see Figure 5-5.

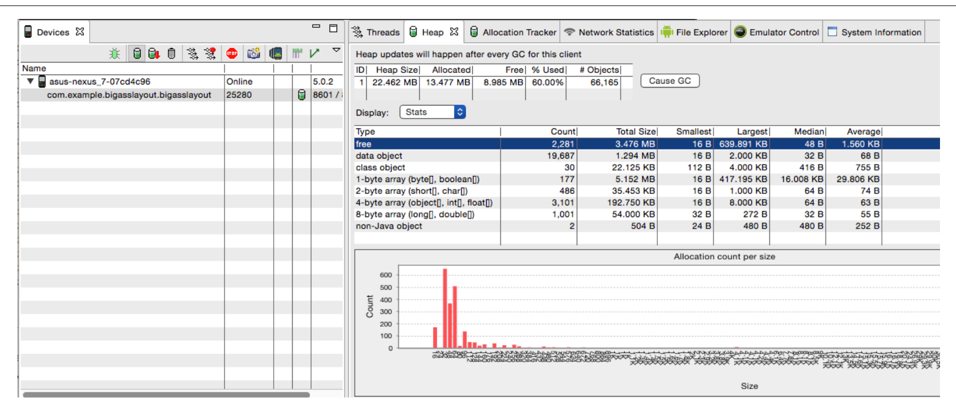


Figure 5-5. Heap dump results for unoptimized app

The Heap tool lists the device(s) on the left, and to the right, includes a table breaking down how memory is allocated. Below the table is a bar chart showing the number of objects by size. We can learn about how memory is allocated in the app by looking more closely at the table (Figure 5-6).

Heap updates will happen after every GC for this client

ID	Heap Size	Allocated	Free	% Used	# Objects
1	22.462 MB	13.477 MB	8.985 MB	60.00%	66,165

Cause GC

Display: Stats

Type	Count	Total Size	Smallest	Largest	Median	Average
free	2,281	3.476 MB	16 B	639.891 KB	48 B	1.560 KB
data object	19,687	1.294 MB	16 B	2.000 KB	32 B	68 B
class object	30	22.125 KB	112 B	4.000 KB	416 B	755 B
1-byte array (byte[], boolean[])	177	5.152 MB	16 B	417.195 KB	16.008 KB	29.806 KB
2-byte array (short[], char[])	486	35.453 KB	16 B	1.000 KB	64 B	74 B
4-byte array (object[], int[], float[])	3,101	192.750 KB	16 B	8.000 KB	64 B	63 B
8-byte array (long[], double[])	1,001	54.000 KB	32 B	272 B	32 B	55 B
non-Java object	2	504 B	24 B	480 B	480 B	252 B

Figure 5-6. Heap dump table: unoptimized app

These are the results for the "Is it a goat?" app with the bloated layout, adding extra objects, and not invalidating the main view (all options available in the settings menu). Just looking at this screen causes a 22.5 MB heap to be created. 13.7 MB of memory is actively allocated to 66,165 objects. We can see how much of that memory is in objects, classes, and arrays in the larger table. Note that images are stored as byte arrays, and this `byte[]` has the most memory allocated to it.

Another interesting feature of this is that the app keeps 40% of the heap size as free, but also keeps a large portion (nearly 3.5 MB) of the allocated memory as free space. If you look at the "free" line that is highlighted, these free allocated spaces are pretty highly fragmented. Of 2,281 free spaces, the smallest free space is 16 B and the largest is 639 KB. However, the median is 48 B, meaning that half (or 1,140) of these allocated free spaces are under 48 B. When new objects are created, only the smallest new objects will fit into these spaces. So it is also important to know what objects you allocate during the runtime of your app.

As you might recall from Chapter 4, the "Is it a goat?" app has various view files that go from unoptimized to optimized. Rerunning the heap dump with the optimized "More Optimized Layout:RL" view while also choosing the options "invalidating the main view" and not "creating extra objects" removes a lot of objects and memory. How much? We can compare Figures 5-6 and 5-7 to find the answer.

Heap updates will happen after every GC for this client

ID	Heap Size	Allocated	Free	% Used	# Objects
1	21.208 MB	12.725 MB	8.483 MB	60.00%	55,266

Cause GC

Display: Stats

Type	Count	Total Size	Smallest	Largest	Median	Average
free	1,911	4.300 MB	16 B	379.703 KB	64 B	2.304 KB
data object	10,576	657.141 KB	16 B	2.000 KB	32 B	63 B
class object	30	22.125 KB	112 B	4.000 KB	416 B	755 B
1-byte array (byte[], boolean[])	177	5.152 MB	16 B	417.195 KB	16.008 KB	29.806 KB
2-byte array (short[], char[])	487	35.578 KB	16 B	1.000 KB	64 B	74 B
4-byte array (object[], int[], float[])	1,880	122.922 KB	16 B	8.000 KB	64 B	66 B
8-byte array (long[], double[])	433	21.188 KB	32 B	272 B	32 B	50 B
non-Java object	2	504 B	24 B	480 B	480 B	252 B

Figure 5-7. Heap dump table: optimized app

The changes in the app were: view hierarchy is much reduced and overdraw minimized. Objects are created at runtime, and not in the code. Also, the views are invalidated, allowing for faster GC on their data.

Let's first look at the total heap size. It drops 1,254 KB (or 5.5%), which will certainly aid performance on lower-end devices. The number of objects is roughly 11,000 lower, mostly in data objects, but also a sizable number of 4 B and 8 B arrays. The 1 B arrays are unchanged at 5.152 MB. Images are stored in 1 B arrays, so each of the 12 thumbnails are stored in this section of the heap. The image memory usage does not change across the two views, as each image is allocated in memory just once (even if they are used multiple times in the view hierarchy).

The heap dump tool categories memory usage by type, but if you want to find memory issues, sometimes you need to go all the way to discrete objects to find memory issues. The Allocation Tracker tool can help with this.

Allocation Tracker

To discover what objects your app allocates during the runtime of your app, the Allocation Tracker in DDMS is a great place to start. Allocation Tracker tracks every object allocated into memory during a stretch of time. This is a great way to see if you are unnecessarily creating objects that might be filling up your memory or blocking rendering.

To collect the list of allocations, press the Start Tracking button. Perform your test, and then click Get Allocations. The list of objects created and allocated into memory during that period will appear in the chart. The allocation tracker tests are cumulative, so if you click Get Allocations a second time without first selecting Stop Tracking, the initial results will be added to the second test. For that reason, I recommend

that you restart the tool for each test. Figure 5-8 is an example of a list of allocations collection.

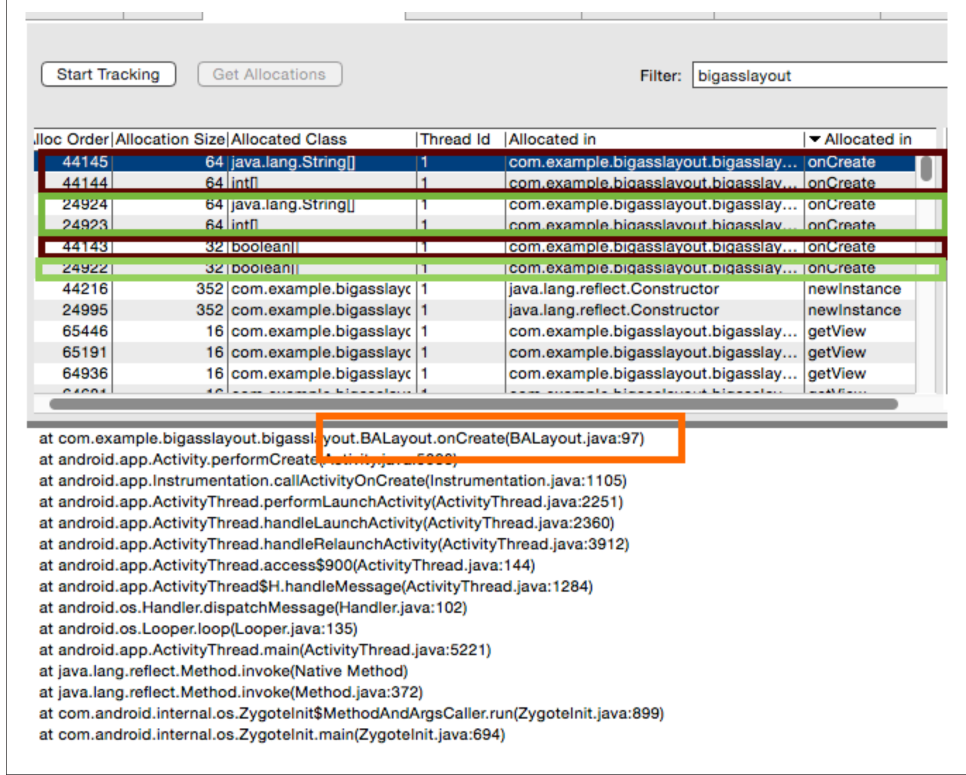


Figure 5-8. Allocation Tracker showing redundantly created arrays

In the test shown in Figure 5-8, I ran the "Is it a goat?" app, and collected all of the allocations from rotating the screen from portrait to landscape and back to portrait. The table allows you to sort by any of the columns, and there is a filtering mechanism. Because the main activity is called com.bigasslayout (recall that there are several large donkey images hidden in the view hierarchy), I performed a filter on the activity name. Digging through the results (through lots of sorting of the columns to find a pattern), I discovered that I was creating three arrays each time I rotated the device (`string[]`, `int[]`, and `byte[]`). These arrays build the views, and don't change, so should have been saved as static or stored in the saved configuration file to prevent their duplication. The larger 44k-byte arrays (in the red boxes) are due to the portrait view displaying more data than the landscape rows (green and ~24,000 B per array). Selecting an allocated item (in this case, the top String array) provides details in the bottom part of the screen. In this case, it shows that this array was generated in line 97 of the of the BALayout code (orange box).

While these three arrays are not a large amount of data, it is a simple example of how creating unnecessary objects adds additional memory requirements (and additional garbage collection), and how removing them will reduce the memory usage of your app. In the “Is it a Goat?” app, you can replicate this report by selecting the Create Objects During Render box in the settings menu. This removes the arrays from the saved configuration, forcing the app to re-create these menus on every rotation of the device. De-selecting this will allow the saved state to be used, and you will not see these three files re-created on every screen rotation.

Adding a Memory Leak

I have added an option in the “Is it a goat?” app that adds a memory leak. Here is what it is doing:

```
//snip

class Iceberg{
    static ArrayList<byte[]> iceSheet = new ArrayList<byte[]>();
    void sink(){
        byte[] mostlyUnderwater;
        mostlyUnderwater = new byte[2048 * 1024];
        iceSheet.add(mostlyUnderwater);//icesheet should grow by 2MB every rotation
        Log.i("iceberg", "Captain, I think we might have hit something.");
    }
}
class CancelTheWatch{
    static Iceberg iceberg;
}
//snip
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
                //snip
        if (memoryLeakTF) {
            //calling the memory leak class
                        // When the Titanic canceled the watch, they hit
                        // an iceberg...
            CancelTheWatch NoNeed = new CancelTheWatch();
            Iceberg theBigOne = new Iceberg();
            NoNeed.iceberg = theBigOne;
                        //this leaks memory.
                        <snip>
                        //next line to quickly run out of memory
            NoNeed.iceberg.sink();
        }
```

There are two things happening here. By calling `theBigOne` from the `Iceberg` class, and then assigning `theBigOne` into the static `Iceberg` inside `CancelTheWatch`, the static class lives longer than the view, so when I rotate the screen, the view cannot be destroyed as the new view is generated and a leak is created.

The `Iceberg` leak shown here is not a huge one. In order to radically inflate the memory heap of the app and see an out-of-memory error, the `Iceberg.sink` object creates a 2 MB byte array (`mostlyUnderwater`) and adds it to the `ArrayList iceSheet`. On devices with lower available memory (in this case, a Samsung Galaxy Note II on Jelly Bean), this can quickly lead to a crash:

```
02-03 02:10:27.650    9399-9399/<app name> D/AbsListView:
                                 Get MotionRecognitionManager
02-03 02:10:31.680    9399-9399/<app name> D/dalvikvm: GC_FOR_ALLOC freed 782K,
        7% free 17078K/18311K, paused 36ms, total 38ms
02-03 02:10:31.680    9399-9399/<app name> I/dalvikvm-heap: Grow heap (frag case)
        to 19.108MB for 2097168-byte allocation
02-03 02:10:31.695    9399-9399/<app name> I/iceberg:
        Captain, I think we might have hit something.
02-03 02:10:31.710    9399-9402/<app name> D/dalvikvm: GC_CONCURRENT freed 611K,
        10% free 18514K/20423K, paused 11ms+2ms, total 27ms
02-03 02:10:31.710    9399-9399/<app name> D/dalvikvm:
                                 WAIT_FOR_CONCURRENT_GC blocked 11ms
02-03 02:10:31.725    9399-9399/<app name> D/AbsListView:
                                  Get MotionRecognitionManager
02-03 02:10:35.440    9399-9399/<app name> D/dalvikvm:
                                  GC_FOR_ALLOC freed 39K, 7% free
        19151K/20423K, paused 18ms, total 18ms
02-03 02:10:35.445    9399-9399/<app name> I/dalvikvm-heap:
        Grow heap (frag case) to 21.132MB for 2097168-byte allocation
02-03 02:10:35.470    9399-9399/<app name> I/iceberg:
        Captain, I think we might have hit something.
02-03 02:10:35.470    9399-9410/<app name> D/dalvikvm: GC_FOR_ALLOC freed 7K,
        6% free 21191K/22535K, paused 24ms, total 24ms<
```

The preceding logs show the memory changes to the app when I rotated the screen twice. The heap grows by 2 MB (02:10:31.680 and 2:10:35.445) after each rotation by 2 MB (to 19 MB and then to 21 MB). There are four garbage collections shown occurring before and after each heap increase. `GC_FOR_ALLOC` occurs to free up memory to make room to fulfill the allocation request. These will cause jank in the app, as they pause the system for 36, 18, and 24 ms each. The `GC_CONCURRENT` is the general GC that runs periodically to clean up objects, and its 11 ms pause might be long enough to cause a jank issue.

I continued rotating the screen, and the memory continued to balloon (as you can see here we are now at 58.5 MB), and the garbage collector is doing everything it can to prevent an out-of-memory error:

```
02-03 02:11:23.125    9399-9399/<app name> D/dalvikvm: GC_FOR_ALLOC freed 659K,
        7% free 57413K/61639K, paused 28ms, total 29ms
02-03 02:11:23.130    9399-9399/<app name> I/dalvikvm-heap:
        Grow heap (frag case) to 58.498MB for 2097168-byte allocation
02-03 02:11:23.145    9399-9399/<app name> I/iceberg:
        Captain, I think we might have hit something.
02-03 02:11:23.160    9399-9402/<app name> D/dalvikvm: GC_CONCURRENT freed 259K,
```

```
        8% free 59202K/63751K, paused 12ms+2ms, total 27ms
02-03 02:11:23.160    9399-9399/<app name> D/dalvikvm:
                                WAIT_FOR_CONCURRENT_GC blocked 14ms
02-03 02:11:23.175    9399-9399/<app name> D/AbsListView:
                                Get MotionRecognitionManager

02-03 02:11:28.480    9399-9399/<app name> D/dalvikvm: GC_FOR_ALLOC freed 36K,
        7% free 59705K/63751K, paused 16ms, total 16ms
02-03 02:11:28.480    9399-9399/<app name> I/dalvikvm-heap: Forcing collection of
        SoftReferences for 2097168-byte allocation
02-03 02:11:28.505    9399-9399/<app name> D/dalvikvm: GC_BEFORE_OOM freed 80K,
        % free 59624K/63751K, paused 26ms, total 26ms
02-03 02:11:28.505    9399-9399/<app name> E/dalvikvm-heap:
        Out of memory on a 2097168-byte allocation.
02-03 02:11:28.505    9399-9399/<app name> I/dalvikvm:
                                "main" prio=5 tid=1 RUNNABLE
02-03 02:11:28.505    9399-9399/<app name> I/dalvikvm:
        | group="main" sCount=0 dsCount=0 obj=0x418b9508 self=0x418a03f0
02-03 02:11:28.505    9399-9399/<app name> I/dalvikvm:
        | sysTid=9399 nice=0 sched=0/0 cgrp=apps handle=1074749232
02-03 02:11:28.505    9399-9399/<app name> I/dalvikvm:
        | schedstat=( 6822358884 1174852496 11100 ) utm=615 stm=67 core=3


02-03 02:11:28.505    9399-9399/<app name> D/AndroidRuntime: Shutting down VM
02-03 02:11:28.505    9399-9399/<app name> W/dalvikvm: threadid=1:
        thread exiting with uncaught exception (group=0x418b82a0)
02-03 02:11:28.510    9399-9399/<app name> E/AndroidRuntime: FATAL EXCEPTION: main
    java.lang.OutOfMemoryError
            at <app name>.BALayout$Iceberg.sink(BALayout.java:77)
            at <app name>.BALayout.onCreate(BALayout.java:234)
            at android.app.Activity.performCreate(Activity.java:5206)
```

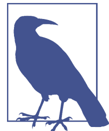
In the log excerpt above, the app memory usage has ballooned to over 58 MB and the device is running out of memory. Let's see what Android does to prevent an out-of-memory crash to your app. The app is attempting to allocate another 2,097,168 B array into memory, and there is no longer any room. First, Dalvik forces the collection to `SoftReferences`, and then we have the `GC_BEFORE_OOM` (the last chance garbage collection before an out-of-memory error), and because these GCs could not find an available 2 MB segment of memory for my byte array, the app crashes.

Now, generally leaks are not easy to find by just looking at the logs, but there are some specialty tools to help you diagnose memory leaks, so you can find their source and resolve them. Let's see how we can identify this leak in the jHat and MAT toolsets.

Deeper Heap Analysis: MAT and LeakCanary

In order to diagnose where your app is leaking memory, you will need to analyze all of the files that your app is holding in memory. If you are able to identify files that should have been released, or identical duplicate files in memory, you can resolve the issue in your code. This might mean ensuring that objects are released properly, or perhaps ensuring that files in memory are reused (rather than having multiple instances stored in memory.)

In order to analyze the files in your app's memory, you'll need to save a memory heap dump to the computer. Next to the Heap Dump icon in Monitor (the cylinder half full of green Android goo) is a similar icon, but with a red arrow pointing down. This allows you to save the heap dump to your computer for further analysis.



The saved heap dump is in an Android-specific format. In order to open the file with other tools, you must convert the file. The conversion tool is hprof-conv, and is included in the Android SDK platform-tools directory:

```
hprof-conv _<existing_filename> <converted_filename>_
```

If you collect your heap dump from Android Studio's DDMS, you do not need to run the conversion because it is run automatically.

When creating your heap dump, try to replicate the steps that cause large memory issues. If you can get your app to balloon in size, or mimic whatever behavior is being reported, the memory data will be in the memory dump hprof file. Leaks can be tricky to find, and might just require a lot of staring at the tool, so the larger the leak is, the easier it will be to find.

To analyze this heap dump, we'll look at Eclipse Memory Analysis Tool (MAT). In early 2015, Square released LeakCanary, an open source library that automates much of the MAT analysis for you, and will report memory leaks in your app while you are debugging. Let's first understand how to find memory leaks using MAT, and then see how LeakCanary simplifies the process for us.

MAT Eclipse Memory Analyzer Tool

Eclipse's Memory Analyzer Tool (MAT) is exactly what the name says: a tool to perform detailed memory heap analyses. MAT is part of the Eclipse IDE, but if you have migrated your Android development to Android Studio, it can be downloaded as a standalone app from Eclipse.org.

When you open your hprof file in MAT, it does some processing of the file, and asks if you'd like a custom report. Because we are looking for memory leaks, I typically

choose the Leak Suspects report. This will show you the objects that are using the most memory. Once these have run, you'll see a number of tabs open in the tool.

The MAT tool provides a wealth of data in a number of different windows. In Figure 5-9, we are focusing on the Overview tab of the main view. It displays a pie chart of the major consumers of memory. Each area of the pie chart represents a chunk of allocated memory, and mousing over each area gives you details about that memory object. The largest chunk of memory is gray, and represents free memory. The second largest class is the `Iceberg` class (as a result of the large `ArrayList` holding two byte arrays), weighing in at 4 MB.

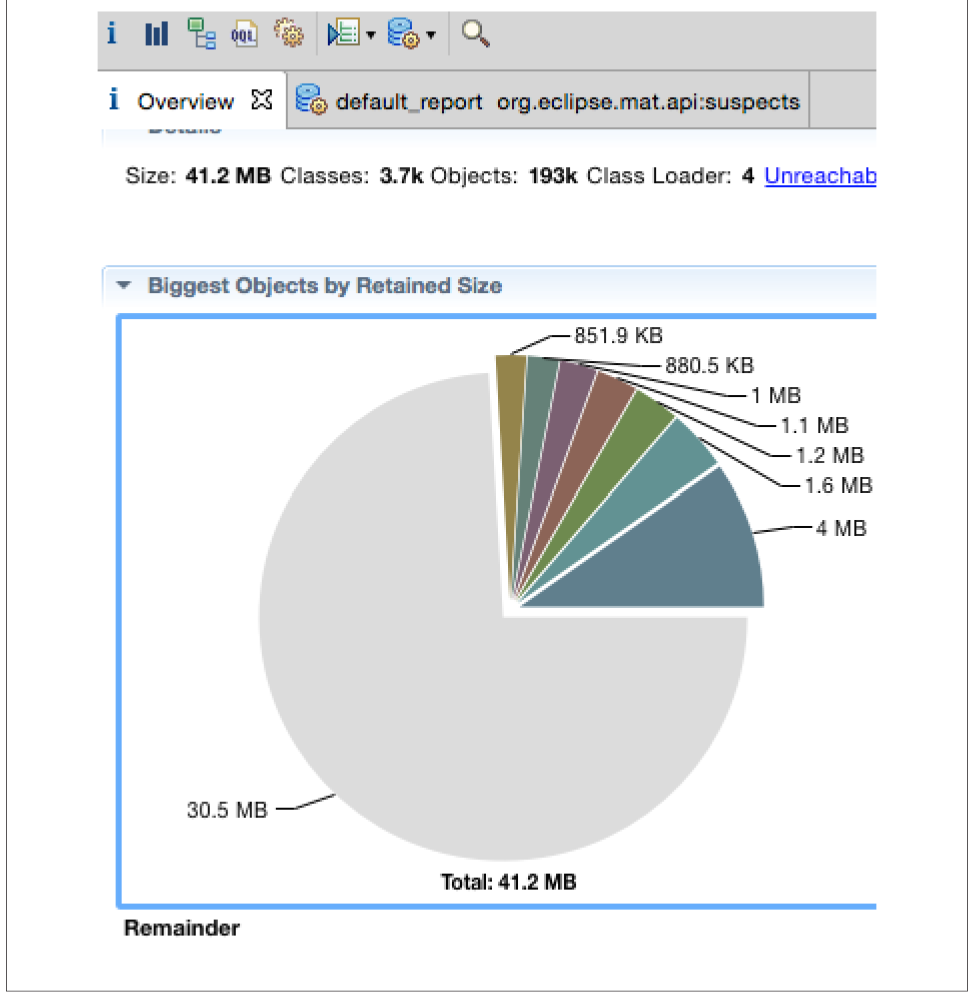


Figure 5-9. MAT Overview

When the `Iceberg` class is highlighted in the pie chart, the Inspector window (Figure 5-10) provides more information about the objects currently referenced by the `Iceberg` class object. As we can see in the code, the `iceSheet ArrayList` is shown.

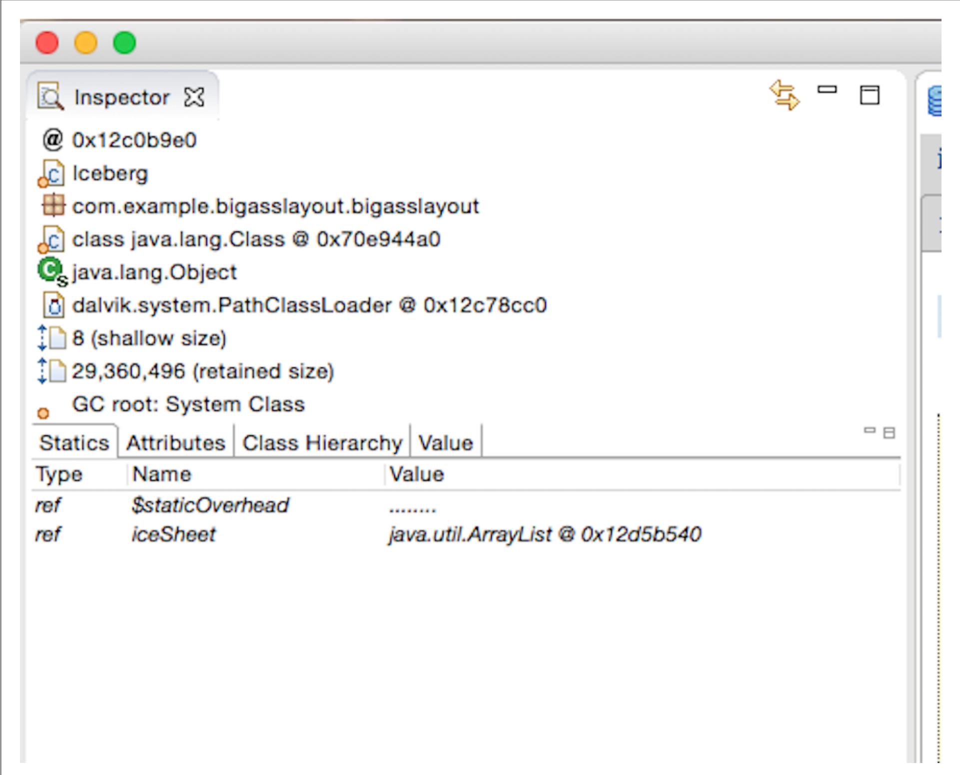


Figure 5-10. MAT Inspector window

Switching the main view tab from Overview to the Leak Suspect report, there is another pie chart listing the suspects (based on memory used). Figure 5-11 shows two pie charts from two separate heap dumps. In the graph on the left, which was run after only two screen rotations, there are two suspects indicated, the larger dark blue section using 27 MB (byte arrays) and the teal green at 6.1 MB (Java classes). The chart on the right was run after many screen rotations, and the memory utilized by byte arrays (now teal green) remains at 27 MB, but the Java class memory allocation (now dark blue) has ballooned to 36 MB. If we had not known already, this looks like a good place to find a memory leak.

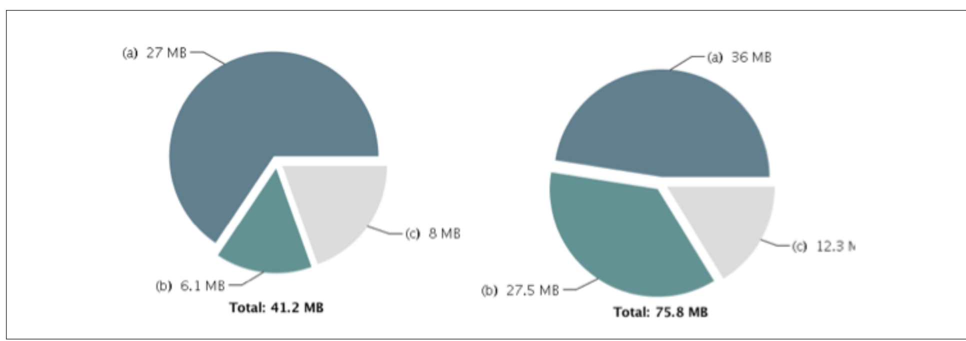


Figure 5-11. MAT leak suspects from two memory heap dumps

Below the pie chart are yellow boxes describing all of the suspects (see Figure 5-12). In this case, we'll continue our analysis with the second trace (run after many screen rotations).

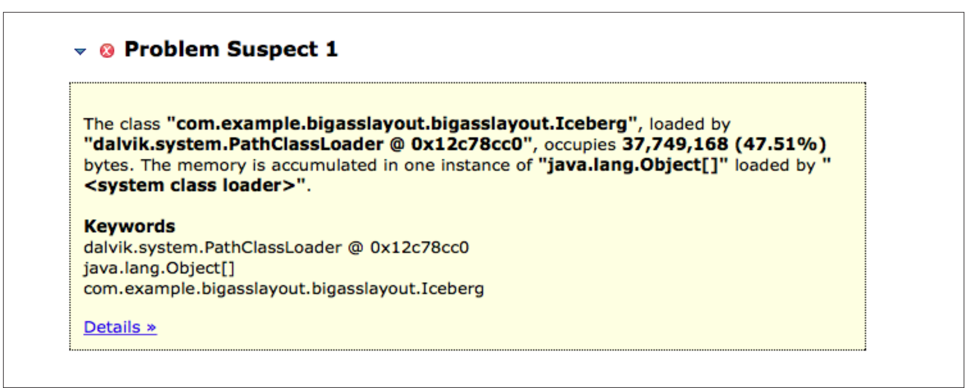


Figure 5-12. MAT leak suspect 1

Suspect 1 is the Iceberg class, using 37 MB (47% of total memory) in one Java object. We can learn more about this suspect by clicking the Details link.

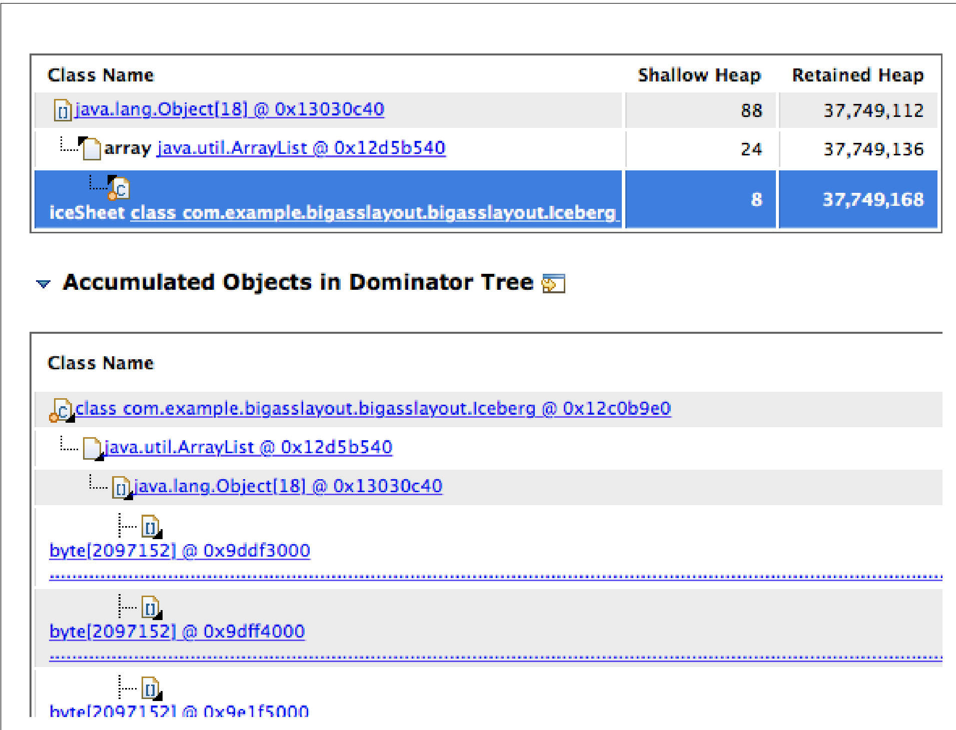


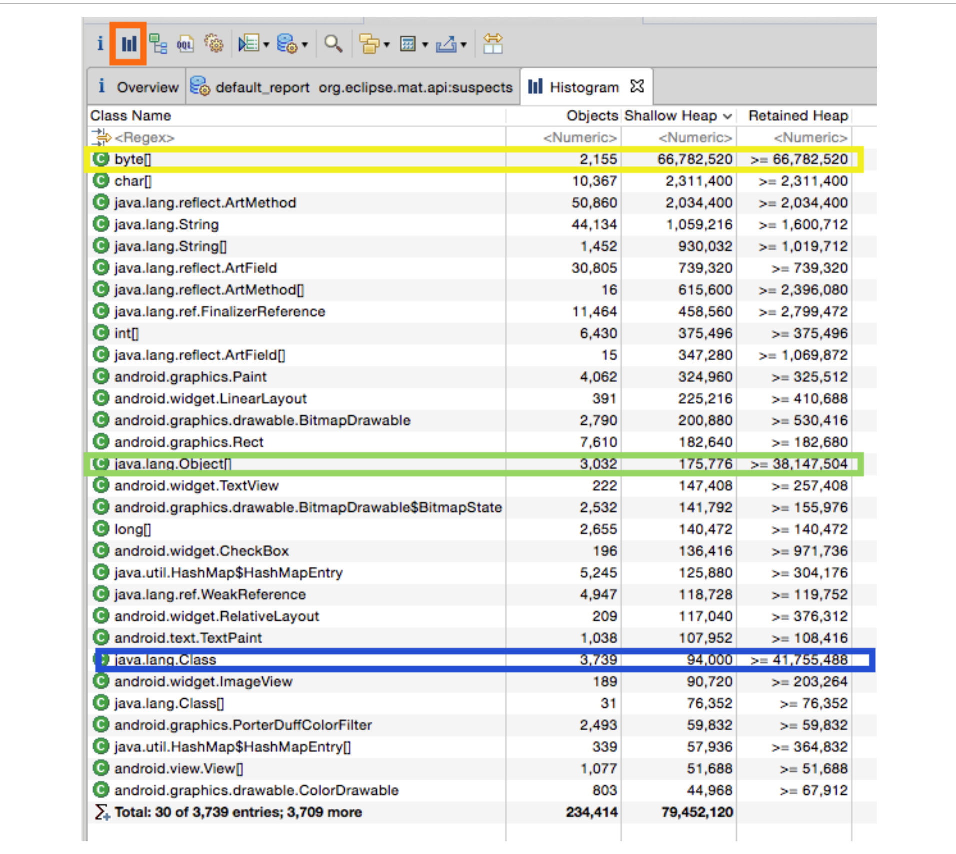
Figure 5-13. MAT Leak View

In this case, the leak suspect report has nailed it. The Shortest Path to Accumulation Point (the path of references to the object keeping this in memory) view pushes us straight to the `ArrayList iceSheet`. Granted, in this sample, the path is not complicated, but it did work.

There is some neat memory information here too: `iceSheet` has a shallow heap of 8 B, but a retained heap of 37 MB. Shallow heap is the memory being taken by just the object, while retained heap is the memory of the object *plus* all of the objects that this object has references to (in this case, 18 2.09 MB byte arrays). Just like a root holds a tree in place, objects that are still in memory hold all other objects they refer to in memory. This is obviously our leak.

It is rarely this simple. If the leak were not so Titanic in size, more digging might be required. Let's look at some of the other options in MAT that you can used to isolate memory leaks.

A memory Histogram is created by pressing the icon that looks like a bar chart (marked by an orange rectangle in Figure 5-14).



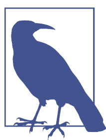
Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[]	2,155	66,782,520	>= 66,782,520
char[]	10,367	2,311,400	>= 2,311,400
java.lang.reflect.ArtMethod	50,860	2,034,400	>= 2,034,400
java.lang.String	44,134	1,059,216	>= 1,600,712
java.lang.String[]	1,452	930,032	>= 1,019,712
java.lang.reflect.ArtField	30,805	739,320	>= 739,320
java.lang.reflect.ArtMethod[]	16	615,600	>= 2,396,080
java.lang.ref.FinalizerReference	11,464	458,560	>= 2,799,472
int[]	6,430	375,496	>= 375,496
java.lang.reflect.ArtField[]	15	347,280	>= 1,069,872
android.graphics.Paint	4,062	324,960	>= 325,512
android.widget.LinearLayout	391	225,216	>= 410,688
android.graphics.drawable.BitmapDrawable	2,790	200,880	>= 530,416
android.graphics.Rect	7,610	182,640	>= 182,680
java.lang.Object[]	3,032	175,776	>= 38,147,504
android.widget.TextView	222	147,408	>= 257,408
android.graphics.drawable.BitmapDrawable$BitmapState	2,532	141,792	>= 155,976
long[]	2,655	140,472	>= 140,472
android.widget.CheckBox	196	136,416	>= 971,736
java.util.HashMap$HashMapEntry	5,245	125,880	>= 304,176
java.lang.ref.WeakReference	4,947	118,728	>= 119,752
android.widget.RelativeLayout	209	117,040	>= 376,312
android.text.TextPaint	1,038	107,952	>= 108,416
java.lang.Class	3,739	94,000	>= 41,755,488
android.widget.ImageView	189	90,720	>= 203,264
java.lang.Class[]	31	76,352	>= 76,352
android.graphics.PorterDuffColorFilter	2,493	59,832	>= 59,832
java.util.HashMap$HashMapEntry[]	339	57,936	>= 364,832
android.view.View[]	1,077	51,688	>= 51,688
android.graphics.drawable.ColorDrawable	803	44,968	>= 67,912
Total: 30 of 3,739 entries; 3,709 more	**234,414**	**79,452,120**	

Figure 5-14. MAT Histogram

This report breaks down the memory usage by class (again by shallow and retained heap). In Figure 5-14, there are a couple of clues that point to the issue:

- `byte[]` (yellow box) includes all images (and the items in the `iceSheet ArrayList`). 66 MB is more than I would expect here, especially because Figure 5-11 shows just 27 MB of byte arrays as images.
- `java.lang.Object[]` (green box) has a low shallow heap, but > 38 MB retained
- `java.lang.Class` (blue box) has a similar low shallow heap, but large retained heap.

These are indicative of small files with large references to other objects. So we should dig further into these classes.



If you cannot find your activity (or a class you are interested in) in the Histogram view, clicking <Regex> in the top row allows you to do a regular expression search on class names.

To examine the byte[] list of objects more closely, right-click the row, and choose List Objects→"With Incoming references." This will give you a new table, which is sorted by retained heap (Figure 5-15).

Overview | default_report org.eclipse.mat.api:suspects | Histogram | list_objec

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
byte[2097152] @ 0xa17bf000	2,097,168	2,097,168
byte[2097152] @ 0xa15be000	2,097,168	2,097,168
byte[2097152] @ 0xa13bd000	2,097,168	2,097,168
byte[2097152] @ 0xa0f0f000	2,097,168	2,097,168
byte[2097152] @ 0xa0bff000	2,097,168	2,097,168
byte[2097152] @ 0x9f5ff000	2,097,168	2,097,168
byte[2097152] @ 0x9f3fe000	2,097,168	2,097,168
byte[2097152] @ 0x9f1fd000	2,097,168	2,097,168
byte[2097152] @ 0x9effc000	2,097,168	2,097,168
byte[2097152] @ 0x9edfb000	2,097,168	2,097,168
byte[2097152] @ 0x9ebfa000	2,097,168	2,097,168
byte[2097152] @ 0x9e9f9000	2,097,168	2,097,168
byte[2097152] @ 0x9e7f8000	2,097,168	2,097,168
byte[2097152] @ 0x9e5f7000	2,097,168	2,097,168
byte[2097152] @ 0x9e3f6000	2,097,168	2,097,168
byte[2097152] @ 0x9e1f5000	2,097,168	2,097,168
byte[2097152] @ 0x9dff4000	2,097,168	2,097,168
byte[2097152] @ 0x9ddf3000	2,097,168	2,097,168
byte[1307600] @ 0xa1d6b000 stx.stw.stw.s	1,307,616	1,307,616
byte[872356] @ 0xa1c96000 ..!...!...!...!...	872,368	872,368
byte[745332] @ 0xa08ea000 8g3.8g3.9h3.9	745,344	745,344
byte[653800] @ 0xaf0e0000 Ch%.Af%.<b&	653,816	653,816

Figure 5-15. MAT Byte[] Objects

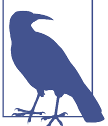
At the top of the list, we can see the 18 2 MB arrays created from rotating the screen. To find the root object blocking these from garbage collection, right-click an object, and select "Path to GC Roots"→ "excluding weak references" (as weak references do not block objects from GC). This will open a new window, as seen in Figure 5-16.

Status: Found 1 paths. No more paths left.

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
byte[2097152] @ 0xa17bf000	2,097,168	2,097,168
[0] java.lang.Object[18] @ 0x13030c40	88	37,749,112
array java.util.ArrayList @ 0x12d5b540	24	37,749,136
iceSheet class com.example.bigasslayout.bigasslayout.Iceberg	8	37,749,168

Figure 5-16. GC roots of 2 Byte[] Objects

The path to GC roots again identifies `iceSheet` as the culprit for our memory leak. I picked the first byte array, and the second line of the report shows that it occupies location [0] in an `ArrayList` that has 18 items in it. The last line names this `ArrayList` as `iceSheet`. We again found our leak! The hprof file is saved in the link to *High Performance Android Apps* GitHub repository. I'll leave tracing the `java.lang.Object` and `java.lang.classes` to the `iceSheet` memory leak as an exercise, but following the same steps will get you the same answer.



If you think the leak is related to an image in a `byte[]` (as all images are stored in memory as byte arrays), but you are not sure what image is causing the problem, there is a way to convert the byte array into an image. The `byte[]` will have a nested object "mbuffer" of class `android.graphics.bitmap`. Clicking this will show the width and height of the object in the Inspector view. Now, right-click the byte array and choose Copy → "Save value to file" (save with extension *.data+*). In a graphics tool like GIMP, you can open this file, apply the height and width values, and GIMP will show you the image hidden in the byte array.

Using the Eclipse MAT to trace how your app allocates memory is a fascinating way to learn about how Android handles memory allocations, and find ways to optimize how your app handles memory. But in the high-speed rush to launch, you might not have time to learn a new tool to investigate difficult to diagnose memory leaks. Luckily for you, the team at Square open sourced LeakCanary, a test tool that automates a lot of what MAT does.

LeakCanary

LeakCanary was developed at Square to reduce the number of out-of-memory errors that they were encountering with their app. They found that replicating crashes involved finding the devices that were crashing, replicating the crashes, and then essentially using trial and error in MAT to find what was causing the leak. Because this approach was slow, they wanted to find the memory leak in their code before launching to their end users. LeakCanary was born. It is the "canary in the coal mine" for memory leaks: it sniffs out memory leaks before any out-of-memory crashes. Since using LeakCanary, Square reports a 94% drop in OOM crashes! Let's see how this tool works!

Square's instructions make it super easy to get LeakCanary up and running. I've implemented it in the "Is this a goat?" app on GitHub.

In the *build.gradle* file, add two dependencies:

```
debugCompile 'com.squareup.leakcanary:leakcanary-android:1.3.1'
releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.3.1'
```

in the `application` class of the "Is it a goat?" app, I added:

```
//LeakCanary reference watcher
public static RefWatcher getRefWatcher(Context context) {
    AmiAGoat application = (AmiAGoat) context.getApplicationContext();
    return application.refWatcher;
}
private RefWatcher refWatcher;


@Override public void onCreate() {
    super.onCreate();
    //on app creation - turn on leakcanary

    refWatcher = LeakCanary.install(this);
}
```

and then I added specific reference watchers for the `CancelTheWatch` and `Iceberg` classes:

```
//LeakCanary watching the variables
RefWatcher wishTheyHadAWatch = AmiAGoat.getRefWatcher(this);
wishTheyHadAWatch.watch(NoNeed);

RefWatcher icebergWatch = AmiAGoat.getRefWatcher(this);
icebergWatch.watch(theBigOne);
```

Now, when I fire up the "Is it a goat?" app, turn on the memory leak, and rotate the screen, a few things happen. After a momentary delay, LeakCanary takes a heap dump and performs an analysis. The report is written to the logs:

```
05-25 15:43:28.283  17998-17998/<app>I/iceberg:
                   Captain, I think we might have hit something.
05-25 15:43:51.356  17998-18750/<app> D/LeakCanary: In <app>:1.0:1.
    * <app>.Iceberg has leaked:
    * GC ROOT static <app>.CancelTheWatch.iceberg
    * leaks <app>.Iceberg instance
    * Reference Key: 52614375-1531-47b1-96d7-4ec986861794
    * Device: motorola google Nexus 6 shamu
    * Android Version: 5.1 API: 22 LeakCanary: 1.3.1
    * Durations: watch=5443ms, gc=154ms, heap dump=2864ms, analysis=14302ms
    * Details:
    * Class <app>.CancelTheWatch
    |   static $staticOverhead = byte[] [id=0x12c9f9a1;length=8;size=24]
    |   static iceberg = <app>.Iceberg [id=0x1317e860]
    * Instance of <app>.Iceberg
    |   static $staticOverhead = byte[] [id=0x12c88e21;length=8;size=24]
    |   static iceSheet = java.util.ArrayList [id=0x12c267a0]
```

The trace is telling me that the `Iceberg` class has leaked, all about the device, how long the processing took (154 ms for the GC, 2 s to collect the heap dump, and 14 s to analyze), and what object in the class caused the leak. The GitHub documentation walks through the steps to report the leak and the heap dump to your servers for aggregation. (Note that this should only be done on debug versions of your app for obvious delay reasons, but is great for internal testing!) Finally, the reports are also shown on your device in the notification bar, and in a new app in your app list called "Leaks" (see Figure 5-17).

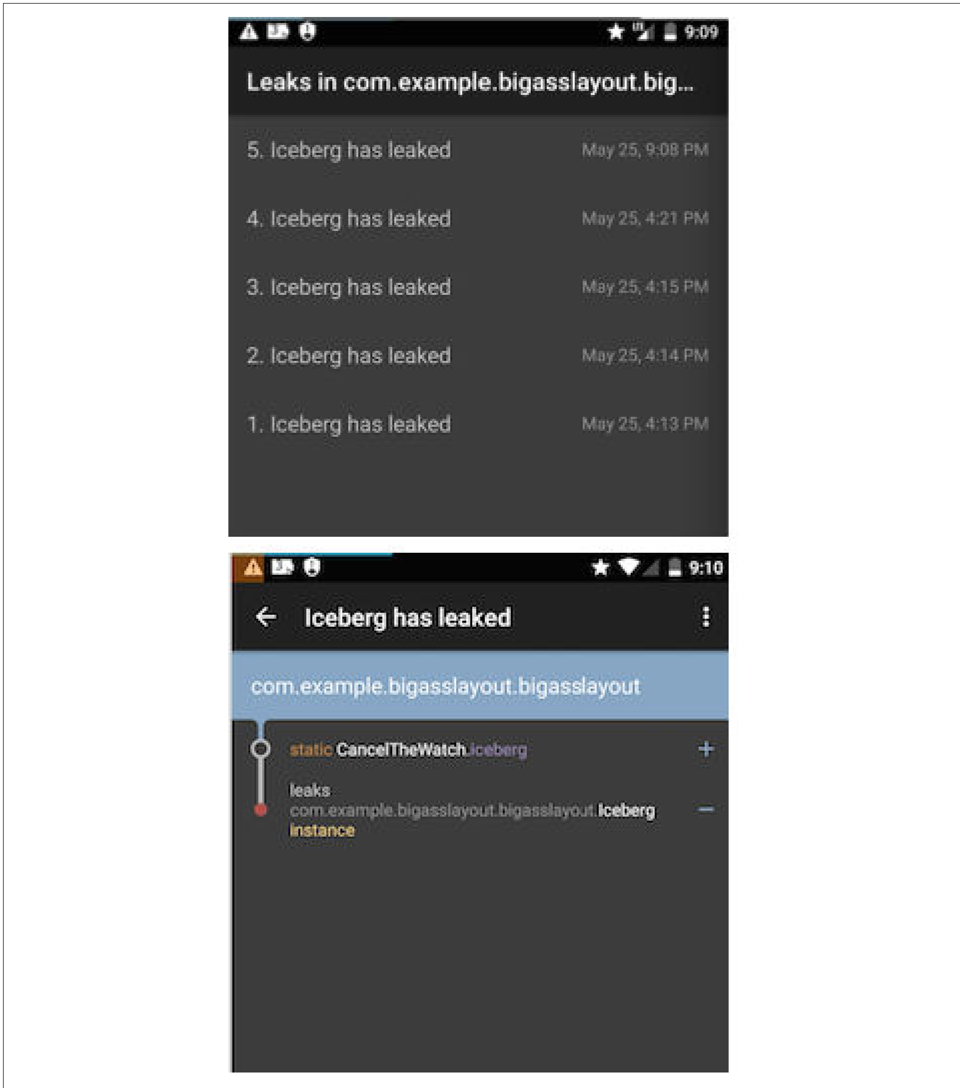


Figure 5-17. LeakCanary screenshots: summary (top) and detail (bottom)

LeakCanary will store the first seven leaks on your device, and has a menu to share the leak and heap dump with others. Using LeakCanary in your internal testing will help you find the memory leak issues that have been eluding you in MAT, helping you quickly squash memory leak issues out of your app, reducing the number of crashes, and improving your app's performance.

Conclusion

Until very recently, the only way to discover memory leak issues was to study all of your out-of-memory crashes, and carefully dig through MAT in order to connect memory reference issues. MAT is still an excellent tool, and it is important to understand the memory linking that MAT exposes. However, the use of MAT in day-to-day testing for memory issues has been alleviated by LeakCanary.

By carefully identifying how your Android app handles memory operations, your app will run more efficiently on memory-constrained devices, and the number of out-of-memory crashes will decline. By limiting your objects and ensuring that their lifespan is appropriate, you can lower the impact of garbage collection on the UI of our app, keeping the GC from blocking the main thread of your app. Finally, by using tools like the Allocation Manager, LeakCanary, and MAT you can identify the objects and classes that are leaking memory.

CHAPTER 6

CPU and CPU Performance

In the preceding chapters, we've looked at battery, UI, and memory management performance, and how optimizing the way these function will reduce crashes and speed up the performance of your app. As we continue our journey to high performance Android apps in this chapter, we'll cover an essential part of the Android device, the CPU. The CPU is the *brain* of the device, and because the CPU processes all of your code to create your app, it is another vital piece of the puzzle to optimize.

In fact, chipset vendors work constantly to improve the performance of their chips, while taking into account battery drain and heat concerns. Modern Android devices have shown great performance strides in speed while also ensuring efficiency.

In the last few years, quad, octo, and deca core CPUs are becoming more common in the market. Unlike your computer (where every CPU is the same, and can be interchanged for any computation), these ARM-based mobile chipsets feature different CPUs for different tasks. ARM calls this chipset design big.LITTLE, and it is a good descriptor for how they work. When a small background task is run (like checking email), a lower-powered, more efficient CPU will be tasked with the job. When users watch videos or play games, the high performance cores are fired up. By relegating small tasks to the LITTLE processors, and only using the big CPUs for high-power tasks, the device saves energy. The great thing as a developer is that this is all controlled by the kernel, and the correct processor will be chosen for you.

As we saw in Chapter 5, even in a memory-managed environment, there are optimizations that we can make to memory. For the same reason, we cannot assume that your app's code will correctly utilize the CPUs on the device. It is still essential to properly administer the way your app utilizes the CPU. In this next section, we'll look at how to understand the CPU usage on your Android device, the CPU usage of your app, and how to determine what threads or processes in your app are causing strain on the CPU. We'll look at how improper use of the CPU can block rendering, or even

cause a dreaded "Application Not Responding" (ANR) warning or even crash your app.

Measuring CPU Usage

Let's start again at a high level and look at how your app may be using CPU in conjunction with the kernel and other apps in the system. The common Linux `top` command is a great way to look at the CPU usage of your app on a device:

```
demo$ adb shell top -n 1 -m 10 -d 1

User 58%, System 14%, IOW 0%, IRQ 0%
User 157 + Nice 6 + Sys 41 + Idle 75 + IOW 1 + IRQ 0 + SIRQ 0 = 280
```

Running the command once (`-n 1`) and getting the top 10 apps using CPU (`-m 10`) over one second (`-d 1`), we can see that 58% of CPU use is user based, and 14% is from the system. The second line tells you how long the scheduler spent in each state (in 10s of ms). The maximum value possible is 100 x the number of CPUS. We see that the active processes account for a total of 280, and as the test was run on a Nexus 6 (with four CPUs), the maximum value is 400.

Now, let's look at the top 10 apps:

```
  PID PR CPU% S  #THR     VSS     RSS PCY UID      Name
15252  1  32% S    16 1581536K  93324K  fg u0_a109  com.example.isitagoat
 1952  0  20% S    97 1708552K 136668K  fg system   system_server
15987  2   2% R     1   4464K   1108K     shell    top
 2413  2   2% S    32 1650148K  76044K  fg u0_a11   com.google.process.gapps
 3010  1   2% S    41 1810248K 179400K  fg u0_a28
                               com.google.android.googlequicksearchbox
 3384  1   2% S    47 1621432K  83928K  fg u0_a11   com.google.process.location
 2586  1   2% S    26 1566872K  93088K  fg u0_a91   com.elvison.batterywidget
 2125  0   1% S    32 1698300K 166068K  fg u0_a24   com.android.systemui
  267  1   1% R    15 227172K  17060K  fg system   /system/bin/surfaceflinger
 6256  1   0% S    49 1603916K  83816K  fg u0_a28
                               com.google.android.googlequicksearchbox
```

As indicated by the table, 32% of the CPU is the "Is it a goat?" app, 20% is the system, the top command takes up 2%, and then a litany of background/Google apps. Running this test while your app is running is a quick-and-dirty way to investigate your CPU usage. We can also see that these apps all have a policy (PCY) of `fg` meaning that they are all visible in one way or other in the foreground.

This is a good start, but we want to get a deeper understanding of the CPU usage of our app. For more detailed information, there is a `dumpsys` command for the CPU:

```
adb shell dumpsys cpuinfo

adb shell dumpsys cpuinfo
Load: 12.28 / 11.64 / 11.56
CPU usage from 11368ms to 4528ms ago with 99% awake:
  0.3% 1531/mediaserver: 0% user + 0.3% kernel / faults: 1093 minor 1 major
  130% 15754/com.coffeestainstudios.goatsimulator: 111% user + 19% kernel /
     faults: 130 minor
  10% 306/mdss_fb0: 0% user + 10% kernel
  9.8% 267/surfaceflinger: 4.5% user + 5.2% kernel
  4.5% 1952/system_server: 1.4% user + 3% kernel / faults: 65 minor
  0.8% 19261/kworker/0:1: 0% user + 0.8% kernel
  0.7% 2982/com.android.phone: 0.2% user + 0.4% kernel / faults: 181 minor
  0.5% 158/cfinteractive: 0% user + 0.5% kernel
  0.5% 18754/kworker/u8:4: 0% user + 0.5% kernel
  0.4% 205/boost_sync/0: 0% user + 0.4% kernel
  0.4% 211/ueventd: 0.2% user + 0.1% kernel
  0.4% 2586/com.elvison.batterywidget: 0.2% user + 0.1% kernel /
  faults: 121 minor
<snip>
```

The first line of the response gives you the average CPU load over the last 1, 5, and 15 minutes. After this, the CPU usage for nearly 7 seconds is shown for all apps (truncated here for space reasons.) For each app, you can see the % of CPU (and if running on more than one core, this can exceed 100%), and the breakdown of this usage between the user and system kernel.

Like most of the other command-line interfaces we have seen, `cpuinfo` is also available as an overlay on your device through the Developer Options (see Figure 6-1). The data is basically the same, but there is an added color bar at the top (underneath the system weighted averages). This shows the time the CPU has spent in userspace (green), kernel (red), and IO interrupt (blue). This can be really helpful to pinpoint the times you might have IO blocking events, as you can see what is on the screen exactly when such an event occurs.

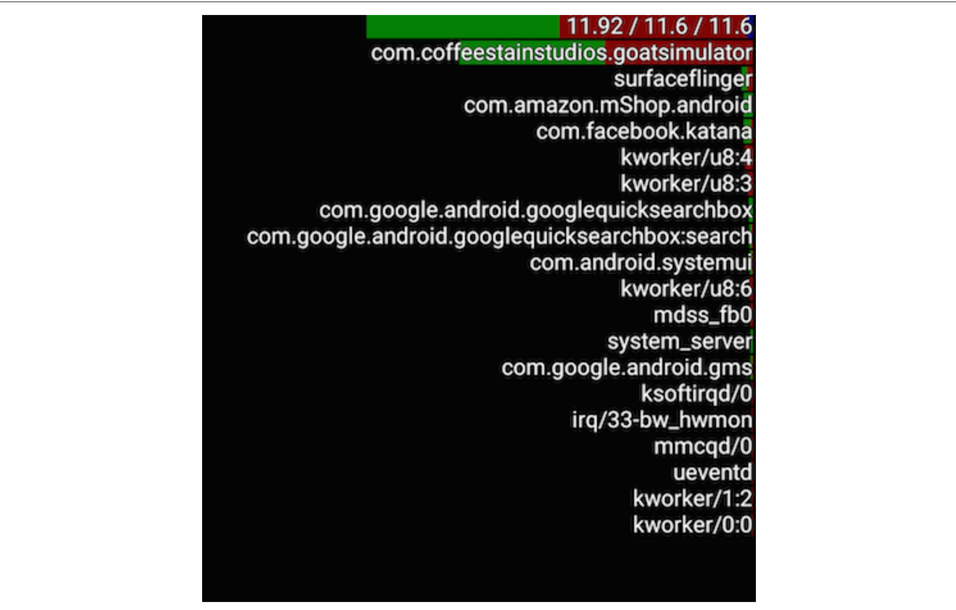


Figure 6-1. Overlay of cpuinfo

Systrace for CPU Analysis

While `top` and `cpuinfo` provide basic understanding on memory usage of your app, we still need to dig deeper into the CPU cores to see what they're actually processing while your app is running. In Chapter 3, we looked at the "Systrace and CPU Usage Blocking Render" on page 113 tool to discover jank in our UI. We can also use Systrace to understand how the CPU can block rendering and cause skipped frames or jank. When we looked at UI, the CPU lines were removed to add visibility. Let's look at them more closely now. The traces described in this section are available in the book's GitHub repository (trace4 has no jank, and trace7 has jank).

At the top of each Systrace (using the same setup as in Figure 4-22), there are rows with information pertaining to each CPU on your test device (see Figure 6-2).

Figure 6-2. Systrace with no jank

When I run a Systrace with the regular views in the "Is it a goat?" app, both CPU0 and CPU1 are in use. There are lots of very small short calculations taking place, but none block the UI. We see very regular creation of views, and the `SurfaceFlinger` sends views to the GPU every 16 ms as we expect. In the two rows of CPU, every colored line is associated with an app. You can identify each process by selecting them and reading the data in the bottom menu, or by zooming in and reading the process name associated with the color (Figure 6-3).

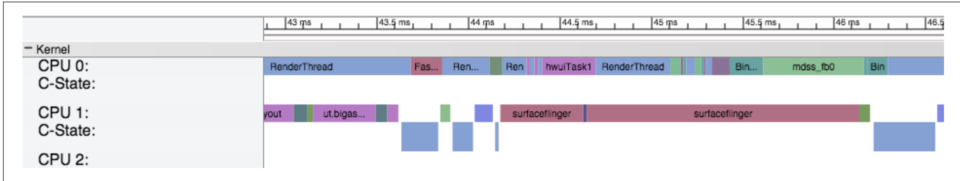


Figure 6-3. CPU view of Systrace

Note that the timescale in Figure 6-3 is a total of 3.5 ms (major ticks are 0.5 ms, and the minor ticks are 0.1 ms). In this very short period, we can see distinct operations (by color):

- Purple is the "Is it a goat?" app
- Blue is the `RenderThread`
- Brick red is the `SurfaceFlinger`
- Many other extremely short processes (lots of which appear as just vertical lines in the current zoom level)

If you look at Figure 6-3 carefully, you will notice that the `RenderThread` and the ut.bigasslayout lines get thicker (take longer) about halfway through of the trace. At this point in the trace, I was touching the screen to change the direction of the scrolling.

In the next systrace, I have turned on the Fibonacci calculation. This calculates a very large number on the 5th and 10th position, causing a large frame in scrolling each time that row is rendered. Figure 6-4 shows a longer duration than Figure 6-3, but fewer views are rendered. For the first 100 ms, everything looks great, no jank and everyone is happy. But about 150 ms in, the UI gets stuck behind the calculation of an eight-digit Fibonacci number. CPU0 (and later CPU2) go solid magenta, as the "Is it a goat?" app is busy doing some serious calculations. The app is stuck on a green obtainView because it is trying to render the view.

Figure 6-4. Systrace with Jank

If you scroll in very closely to the long green obtainView seen at the bottom of Figure 6-4, there is a very thin line with different colored sections right above. In Figure 6-5, we have zoomed into a 15 ms section of the trace, and the thin lines above the green obtainView (bottom row) are dark green and blue indicators. These indicators are telling you what state the CPU is in for your app. The small blue moments are when the process is runnable (but not running), and the green indicates the app is running on the CPU. Drawing vertical lines on the trace shows that the running times coincide exactly to the time that one of several adjacent magenta processes is running on the CPU. The Systrace is showing us that hundreds of small processes running during the obtainView are blocking the app from updating the screen. In this case, the thread that draws the UI is blocking, but it could be possible that a more complicated app could have another thread block the UI rendering.

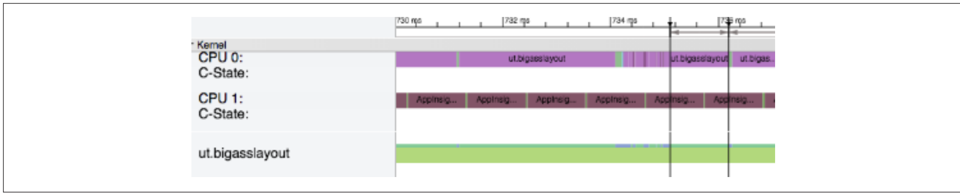


Figure 6-5. App CPU State in Systrace

With the knowledge that there is a process blocking your rendering, now we can apply the Traceview app to further diagnose the problem. There are two incarnations of Traceview, and both display the same information differently. It's worthwhile to discuss both tools, as one version might help you more than the other.

Traceview (Legacy Monitor DDMS tool)

If you have ever watched a video on Android CPU optimization, this is the tool that is typically shown. It has been around from the beginning, and it still incredibly useful. For users of Android Studio, it is most easily accessed from the Monitor tool in the SDK. To start the tool, choose your app, and press the icon that has three horizontal lines with white dots (and one red dot) (inside a red box in Figure 6-6). This will open a box offering you two options for the trace. The first option is to sample all of the processes the VM is running on the CPU every *x* ms (defaulting to 1,000 µs). This is best for devices that are CPU constrained, or if you are planning to take a longer trace. In the examples here, we have chosen the second option, where every method's start and stop is processed. This has higher overhead, and will add latency to apps on even the most powerful devices (the traces in this section were run on a Nexus 6 running 5.0.1).

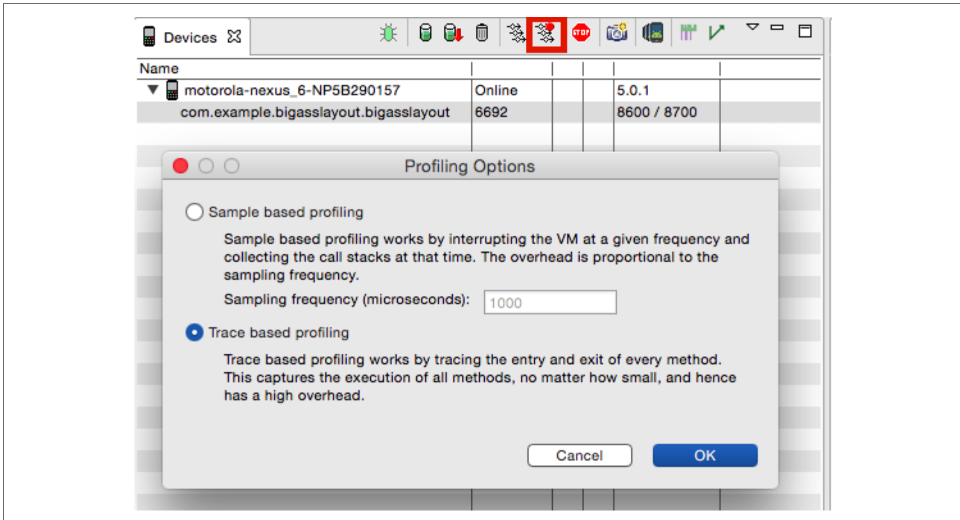


Figure 6-6. Starting Traceview

Once you have started Traceview, run the operations in your app that you would like to test, and stop the trace by pressing the same button you used to start the trace. After a few seconds, a trace will open in the middle window of the DDMS view. Each thread will have a row in the top section (in the case of the "Is it a goat?" app, there is

just the main thread). Each method is shown in a different color (and fully zoomed out, the start and stops all appear black). See Figure 6-7.

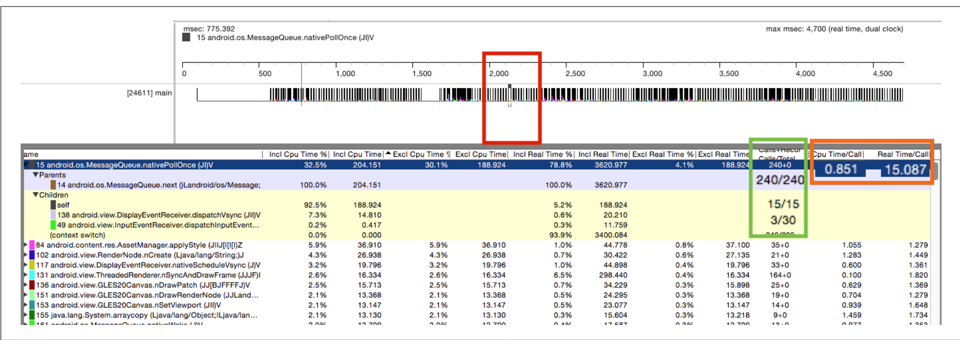


Figure 6-7. TraceView overview

In the table below the Traceview in Figure 6-7, you see a list of all of the methods. Each method can be opened to see its parents and children. In Figure 6-7, I have run the "Is it a goat?" app running in a normal manner (no issues). I have highlighted method 15 (`android.os.MessageQueue.nativePollOnce`) to show that it has the parent `MessageQueue.next`, and two children to dispatch `DisplayEvents` and `InputEvents`. The table lists various breakdowns of how the methods have used the CPU:

Inclusive CPU Time
: Time spent in this method *plus* time spent in child functions

Exclusive CPU Time
: Time spent *only* in this method

Inclusive Real Time
: This is real time (versus time just time utilizing the CPU)

Exclusive Real Time
: This is real time (versus time just time utilizing the CPU)

Calls + Recursive Calls/Total Calls
: The number of times these methods were called in the trace

CPU Time/Call
: Average CPU time per call

Real Time/Call
: Average real time per call

The Calls column tells us how many times each method was called (highlighted in the green box). Method 15 was called 240 times. It calls method 138 a total of 15 times (and is the only parent of this method). It calls method 49 a total of three times (and

other methods call this 27 additional times). Method 15 uses the CPU exclusively for 188.924 ms, and the inclusive time is 204.151 (as method 138 uses 14.8 ms when called). The average time per call on the CPU is 0.851ms, but 15.087 ms in real time (as seen in the orange box).

When you highlight any row (in this case, line 15), each time the method is called, it is highlighted in the traceview above. Alternatively, if you click a region of the graph where method 15 was called, you'd see that region highlighted. At ~2,100 ms in Figure 6-7, you can see one such call in the red box. Traceview denoted the method by adding a solid dark gray bar above and a bracket below the call highlighted. As this method is a part of rendering the view, it is good that this method generally takes < 16 ms to complete. Scrolling in to a 500 ms range, we can see that this method is called for each frame render (Figure 6-8 highlights every instance of method 15).

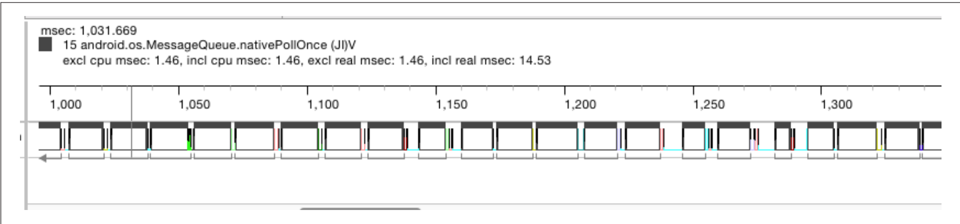


Figure 6-8. Traceview zoomed in showing a highlighted method that reoccurs every 17 ms

Now that we have seen how an app should behave in traceview, let's look at a Traceview of the "Is it a goat?" app with the Fibonacci counter turned on. As shown in Figure 6-9, the difference is immediately apparent.

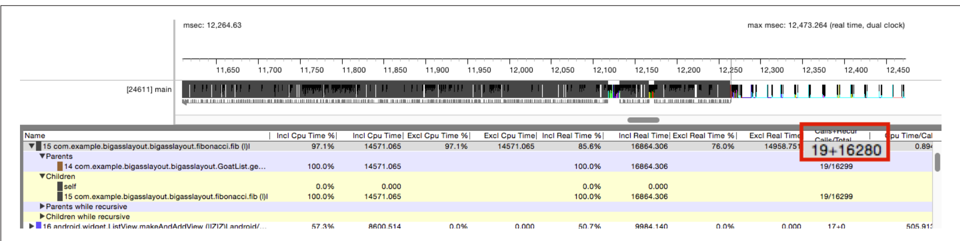


Figure 6-9. Traceview of the "Is it a goat?" app with Fibonacci calculation

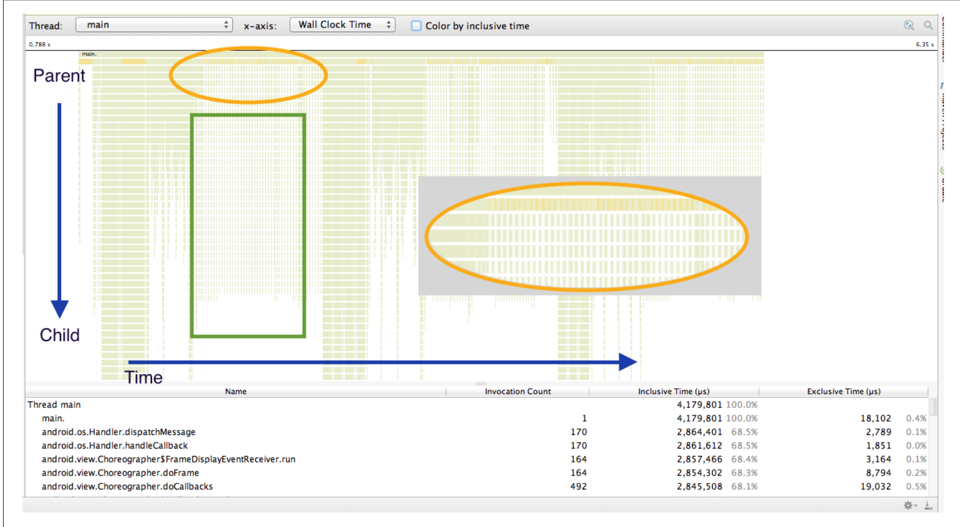
From 11,600–12,250 s, the recursive Fibonacci calculation has completely taken over the main thread, and the black lines in the traceview have become extremely dense. In this case, I have highlighted the Fibonacci process, and each call is highlighted in Figure 6-9. Just as we saw in the systrace, this call blocks nearly every other method in the app. From 12,250–12,450 s, we return to what we would like to see—regular 16 ms cycles on the main thread—indicating a jank free experience.

The table below the TraceView tells us that the Fibonacci method is called 19 times, but because it is a recursive statement, we see that it calls itself another 16,280 times during the trace (enlarged text in red box). The entire trace is over 19 s, and nearly 17 s are spent in this method alone. If we really needed to provide a Fibonacci number to the data, a faster or less CPU intensive method should be applied.

Traceview (Android Studio)

A new Traceview was released with the 0.2.10 release of Android Studio, with the goal of replacing the DDMS/Monitor traceview described in the previous section. The new Traceview uses flamecharts to display the same traces in a different manner. In the Monitor version of Traceview, you can see the direct parent-child relationships of a method, but grandparent-grandchild (or other deeper connections) are difficult to discern without really digging into the table. The flamechart shows you the amount of time each method or process takes along the horizontal axis, but places the process in it the overall parent-child hierarchy on the Y axis. This allows for a deeper visualization into how the method's calls interact.

Running a trace inside Android Studio is easy. Instead of the icon with the red dot (like in DDMS), the icon is a stopwatch. To start Traceview, click the stopwatch. Run your trace, and then hit the stopwatch again to stop (there is no option to change the sampling in the new version.) The Traceview will open in the main view of Android Studio. Immediately, we can see that things are *very* different (Figure 6-10).



Name	Invocation Count	Inclusive Time (µs)		Exclusive Time (µs)	
Thread main		4,179,801	100.0%		
main.	1	4,179,801	100.0%	18,102	0.4%
android.os.Handler.dispatchMessage	170	2,864,401	68.5%	2,789	0.1%
android.os.Handler.handleCallback	170	2,861,612	68.5%	1,851	0.0%
android.view.Choreographer$FrameDisplayEventReceiver.run	164	2,857,466	68.4%	3,164	0.1%
android.view.Choreographer.doFrame	164	2,854,302	68.3%	8,794	0.2%
android.view.Choreographer.doCallbacks	492	2,845,508	68.1%	19,032	0.5%

Figure 6-10. Flamechart Traceview overview

In original traceview, mapping children and parents of methods was done in the table. Here, the parent methods are at the top, and each child method is below it. The horizontal length of each method indicates how long each method was called. Each thread is mapped separately (accessible from the dropdown menu at the top). Threads are colored red, orange, yellow, and green (from slowest to fastest). By default, this is set to the exclusive time, and there is a high-level method (in the second row) that is orange. This is the `MessageQueue.next` method. This method has a large number of calls as it is queueing up views, and it waits for each view to be drawn. The inset orange oval is an enlarged view of the smaller orange oval. It highlights the root methods for a series of regular methods, with a large number of dependencies (green box). These regular calls are animation renders for the bounce animation that occurs when you reach the end of a list. The zoomed area shows that the orange-tinted `MessageQueue.next` runs in between each animation frame.

The `GoatList` method draws each row in the "Is it a goat?" app. It is easy to quickly identify `Goatlist` in the flamechart by using the search function. In Figure 6-11, the rows highlighted in blue denote where the `GoatList` method is called (there are eight instances shown in the figure).

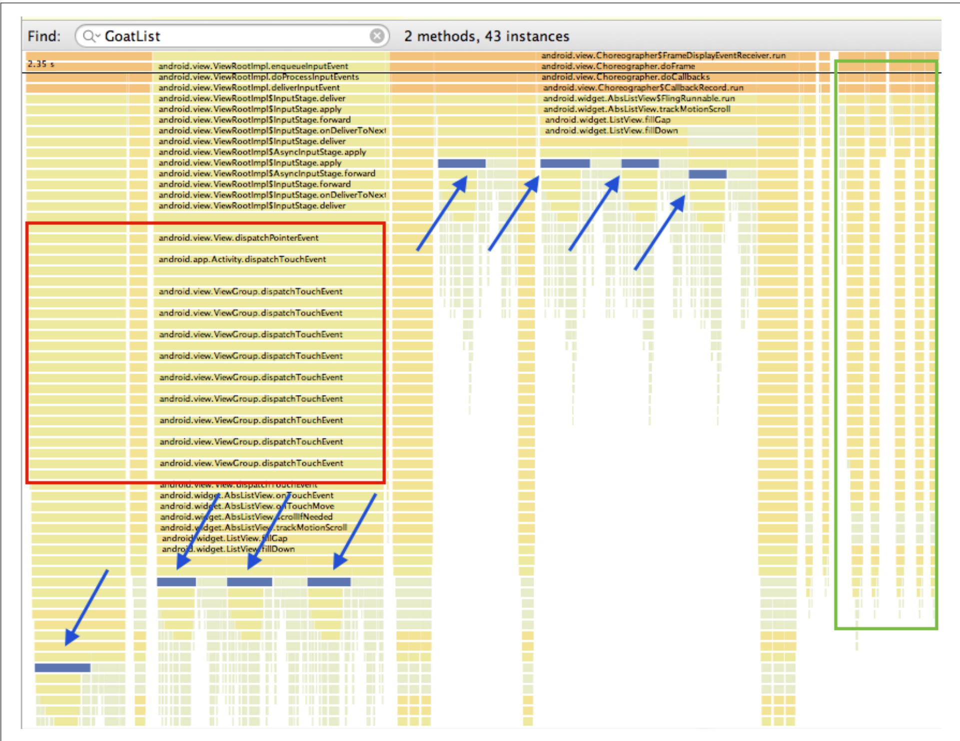


Figure 6-11. Android Studio Traceview GoatList filter

Looking at Figure 6-11, it is interesting to see that `GoatList` appears to be called in two different contexts (based on the Y axis position). This trace was generated in "Slow XML" view, by flinging the view from the top to the bottom. Four of the `GoatList` views are created while the touchevent (calls shown in the red box) is initiating the "fling." The last four `GoatList` rows are created during the rapid fling that occurs after my finger is removed (in the center of the graph). Once the view reaches the bottom, the beginning of the bounce animation can be seen in the green box.

In Chapter 4, we used Hierarchy Viewer to explain the importance of a flat view Hierarchy. We can do a similar analysis in Traceview. The two screenshots in Figure 6-12 are of "Slow XML" above the most optimized layout. The graphs have the same vertical time scale, and it is clear that the bottom view (the more optimized layout) inflates the views faster (26 ms versus 40 ms). Each item takes time, and the number of vertical stalactites is higher in the Slow XML view. There are interesting similarities though. The rendering of the top view is similar for both layouts, and the flamecharts have similar pattern or shape (in the red box). The longest part of this view creation is the addition of the checkbox (orange box), as it has two possible states and an animation when the states toggle.

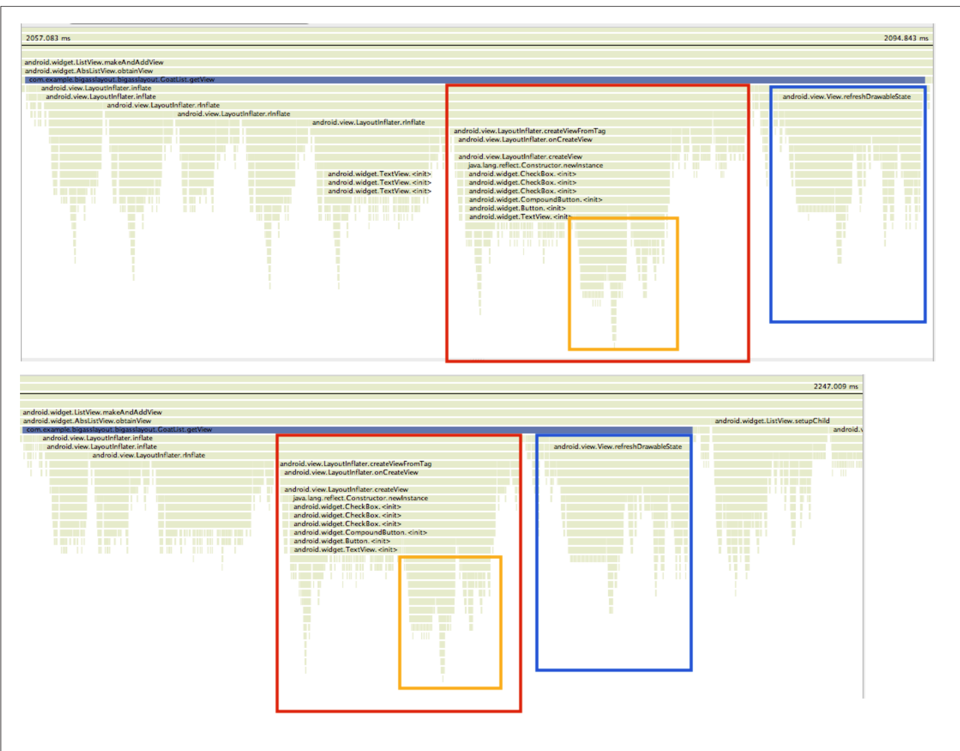


Figure 6-12. Android Studio Traceview GoatList Comparison top: Slow XML view bottom: most optimized view

After the GoatList row is rendered, there is one more set of commands that must be run (shown in the blue box). These methods are required to add a check to the checkbox. For rows in the app that are not a goat (and therefore the checkbox is unchecked), this 5–6 ms is not present. In the app, there are 10 checked rows (they are goats), and three that are unchecked (not goats). When I originally wrote the app, I had all 13 boxes default to checked, and then unchecked the three rows that are not goats. However, I discovered in Traceview that changing every checkbox to checked (and then unchecking programmatically) added the "checked time cost" to every checkbox created, actually adding time to the GoatList layout. As a result of my Traceview findings, I modified to only check the rows that had goats (and required the check).

Now, let's look at what happens when I turn on the recursive Fibonacci calculation, as shown in Figure 6-13.

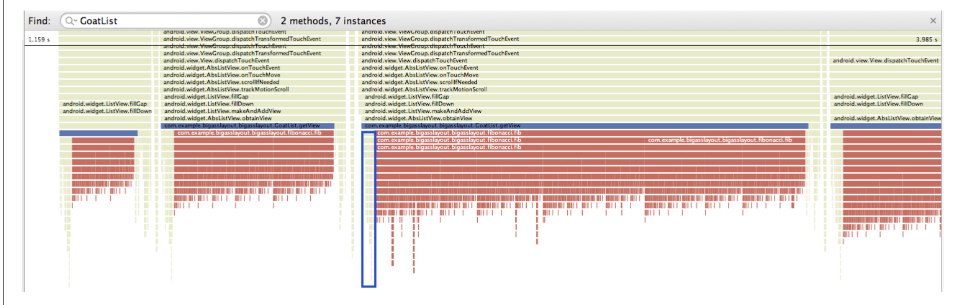


Figure 6-13. Android Studio Traceview GoatList Fibonacci delay

This was actually a tough trace to take, as the overhead from Traceview and the overhead from the recursive Fibonacci calculation caused "not responding" errors to come up on my device. Looking at Figure 6-13, each grouping (there are 3+ shown) is a row being drawn (the GoatList method is highlighted in blue across the top of each grouping). The views inflate as expected at the left of each GoatList, (one example is highlighted in a blue box), but then the required Fibonacci calculation grinds the GoatList method to a halt as it runs its calculation (and is denoted in red as Traceview recognizes that these calculations are causing a slowdown in the app).

When testing across multiple threads, the original Traceview makes it easy to compare what is happening on all threads at any given time. However, the superior flame charting capability in the Android Studio version of Traceview adds significant visualizations as to what is controlling the CPU at any given time in your threads.

Other Profiling Tools

Qualcomm has a free app called Trepn that allows you to show memory, CPU, battery, network, and other characteristics as an overlay on your screen while you test your app. If your phone uses a Qualcomm processor, you can also observe the GPU usage while testing. The data from your traces can be exported into a CSV or database for later analysis. However, each report is calculated individually, so quick comparison of data in the CSV is not simple—loading into your favorite analysis tool is the best approach.

Figure 6-14 is another great way to visually see the CPU usage of your app.

Figure 6-14. Trepn profiling CPU in overlay

Conclusion

The tools described in this chapter are all free to use and provide a great deal of information to developers working to debug memory and CPU issues in their apps. Reducing the CPU footprint of your app allows the `SurfaceFlinger` and framebuffers to ensure 16 ms frame updates and a jank free experience. Reducing the CPU footprint of your app will also save memory and battery, which will in turn speed up the app and reduce jank.

CHAPTER 7
Network Performance

One of the greatest aspects of the smartphone revolution is the ability to tap into a repository of all human knowledge with a small device that fits in your pocket. It allows us to resolve the important questions we may be asked ("Dad, what sound does a giraffe make?"), and it lets us play chess and other games with complete strangers from all around the world.

As demands for network throughput increase, we hear about how faster, more reliable networks will place all of this information closer to your fingertips. I am here to burst that bubble. While newer, faster networks are coming, it will take decades for existing 4G networks to become ubiquitous worldwide. In the meantime, we can focus on how apps use existing networks today, and how important network usage is in relation to your app's performance, but also how it affects the device's battery. As we determined in Chapter 3, the cellular, Wi-Fi, and Bluetooth radios that facilitate all of this amazing communication are also major factors in battery drain. By maximizing your app's network performance, you can make it run significantly faster and use less battery at the same time.

In this chapter, we'll look at the differences between the different data radios on mobile devices, the tools to profile your app's network usage, and some simple fixes that will gain huge improvements. We'll look at how to test your app for different network environments (as much of the world has only 2G and 3G coverage, you should ensure your app performs well under these conditions), and finally will look at the "other radios" of your device: Bluetooth communication with watches/peripherals and GPS location scanning. Let's start by quickly looking at how these radios work, and then describe ways to optimize their use.

Wi-Fi versus Cellular Radios

Wi-Fi versus cellular? Isn't a connection to the Internet just a connection to the Internet? In reality, the ways that these two radios connect are vastly different, and depending on how much data your app requires, you may actually want to architect two different models for content download: one for cellular and one for Wi-Fi.

When connecting to the Internet, there are two aspects to the connection that cause performance constraints: bandwidth (the size of the "pipe") and latency (the length of the "pipe" or how crowded the "pipe" is). We'll look at how these are affected by the various radio connections, and how, despite similar power drain values in the Android Power profile (covered in "Android Power Profile" on page 30) cellular radios use more power when active than Wi-Fi.

Wi-Fi

Wi-Fi connections (in ideal conditions) are high throughput, low-latency connections and are generally unmetered (meaning that there is no additional cost to utilizing Wi-Fi networks). The reason I added the "ideal conditions" waiver to this description is that you are seldom in ideal Wi-Fi conditions. Because Wi-Fi networks utilize the same frequencies, areas with multiple Wi-Fi networks overlap on the limited number of frequencies, resulting in shared bandwidth across all of the networks.

However, let's assume you have a Wi-Fi connection with no bandwidth issues, and a strong connection to your Android phone. When your app attempts to make a Wi-Fi network connection, there is minimal latency to set up the connection. When the connection is established, the radio is on high power. Once the data is transferred, the radio turns off. There is a bit of latency to turn on and turn off the Wi-Fi radios (measured at 80 ms to turn on, and 240 ms to turn off.) As we saw in "Android Power Profile" on page 30 in Chapter 3, Wi-Fi connections on the Nexus 6 use 3 mA of current to stay on in standby mode, and when actively transmitting data they utilize 240 mA. With the limited power of Android devices, you can see why getting content downloaded quickly and efficiently is paramount.

With high throughput, low latency, and no charge for data on Wi-Fi, your app can behave in a more "data hungry" way on Wi-Fi. You can serve higher-quality images and videos, and perhaps have a more interactive experience with your users.

Cellular

There are a variety of different cellular technologies in use around the world today. Depending on what network your customer connects to, the experience could be completely different. As discussed in Chapter 2, many people around the world still utilize 2G and 3G networks. See Table 7-1.

Table 7-1. Evolution of wireless generations

Name	Gen.	Down Max (Kbps)	Latency (ms)
GPRS	2G	237	300–1,000
EDGE	2G	384	300–1,000
UMTS	3G	2,000	100–500
HSPA	3G	3,600	100–500
HSPA+	3.5G	42,000	100–500
LTE	4G	100,000	<100

When connected to a cellular data network, the amount of power your Android device uses to maintain the connection will vary based on signal strength. In "Android Power Profile" on page 30, you may have noticed that there are two values for `radio.on`. If you are in a region of strong cellular signal, you'll use the lower of the numbers, but to maintain a cellular data connection in areas of low coverage, the phone will crank up the power of the antenna to maintain the connection. When the radio is connected, the current jumps from 4.5 mA to 125 mA. While it might appear from these raw numbers that active cellular radio (at 125 mA) uses less power than Wi-Fi (at 240 mA), the power drain for cellular connections is typically higher because of the way cellular connections are implemented on the network. In order to maximize the quality of service on cellular networks, all carriers have implemented a Radio Resource Control (RCC) state machine that controls how data connections are established and taken down.

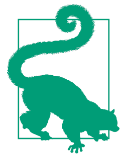
State Machines

A state machine (sometimes a finite-state machine) describes a logical sequence of events with a finite number of states. A simple state machine is a light switch with states off and on. In the case of mobile cellular networks, there are a number of different states that the network connection can have, and they are used in order to optimize several factors, including network and device throughput, latency, and device battery drain.

RRC State Machine

When your phone initiates a data connection, there are several initial radio signals sent to the tower before the TCP connection is established. These signals add an additional 500–1,000 ms latency to the time establishing a radio connection. This latency and delay is one of the crucial aspects of mobile connectivity that the cellular RRC state machine attempts to counteract.

Every mobile network has a RRC state machine that keeps the radio on after the last packet of data is sent in order to offset connection setup latency and also balance power consumption. Each carrier can specify various states, and the time a device remains in the various states (and thus, each network is slightly different). Because each network around the world has different specific variables, the exact timings are not important to know, but a firm grasp of the basics of the RRC state machine is important for understanding how cellular connections operate. Knowing this, you can optimize your connections to work in concert with the RRC state machine, helping to make your mobile app faster and use less battery.



State Machine Caveats

It is well beyond the scope of this book to discuss (or list) the timers across all cellular carriers (and the timers may vary by region even inside a carrier, and will change over time). As a developer, it is not feasible to optimize all of your app connectivity to every carrier's RRC state machine. What is crucial is to understand what a state machine is, and how the existence of a state machine can harm (or help) your mobile app.

Additionally, state machines are different for 3G (GSM, CDMA) and LTE networks. For simplicity, I'll describe the LTE RRC state machine.

4G (LTE) State Machine

When data is to be transmitted, your Android phone goes from an idle state (low power drain) to the connected state (at high power). When the packets have stopped being transmitted, the radio does not immediately shut off. Instead, it enters a high-powered tail time for 10–15 s. If the radio had immediately shut off, data packets sent in quick succession would have to surmount the high latency connection time again, making the user experience incredibly slow. With the radio connection already established, the latency for subsequent packets drops, and the packets are delivered quickly. If no packets arrive during the tail time, the radio closes the connection and shuts down to save power.

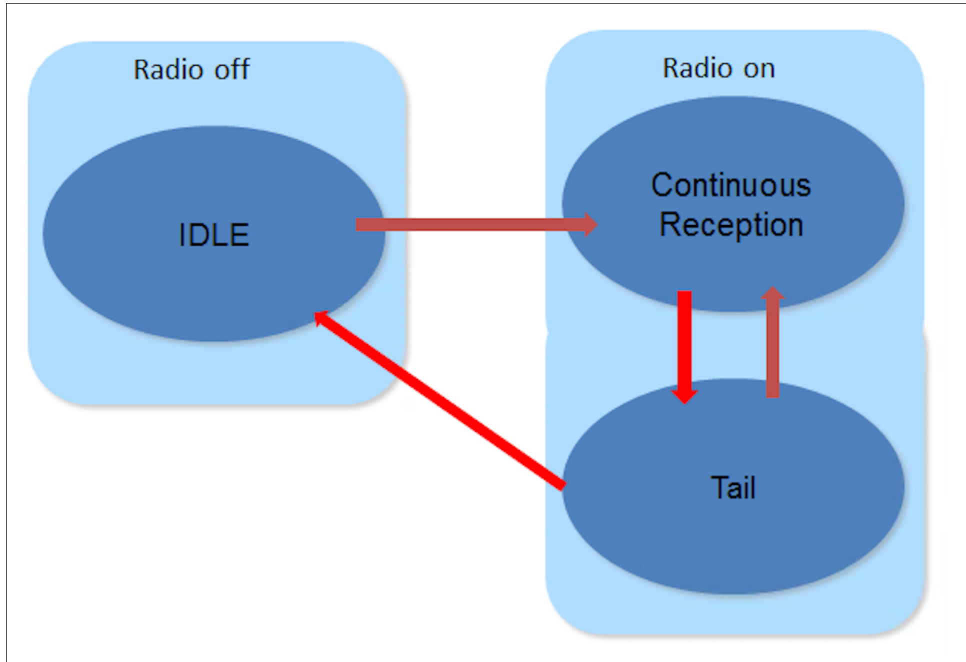


Figure 7-1. The LTE RRC State Machine

If you take another look at Table 7-1, you can see that as network generations improve, the latency decreases and the throughput of the network increases. The 4G network spec has greatly improved the signaling required to establish a data connection (the RRC IDLE → Connected transition), reducing the connection setup latency by a factor of 5–10 from 3G networks (as seen in Table 7-1). What can be 300–1,000 ms on 3G is now 50–100 ms on LTE. While the improvement in bandwidth in 4G is also impressive, it is the latency improvement that really helps LTE feel as fast as it does. Ilya Grigorik covers the effects of latency in great deal in his book *High Performance Browser Networking* (O'Reilly). And he also covers all network performance in much greater detail that we can here.

In general, LTE radios use more power than radios with only a 3G connection. If you are streaming a large file, it is possible that the higher download speeds of LTE will allow radio to complete its connection faster, and use less energy as a result. Most mobile data consumption is not large files, but built of hundreds of smaller files, using smaller chunks of data. These small files are not able to use the full bandwidth capabilities of LTE (because they are so small). So, generally speaking, downloading content over LTE will use slightly more power than on 3G.

Radio Connection Versus Data Connection

There is a nuance that exists between the radio connection and the data connection, and it bears discussion here. The physical radio connection between the tower and the phones is not the same as the data connection that exists between the phone and the server. The data connection travels on top of the radio connection, and as such, a radio connection must be established before data can be transmitted. Think of the radio connection as a lift bridge, and the data connection as the road on that bridge.

If the data connection is left in a connected state for future transmissions, but is not actively transmitting data, the radio connection between the phone and the tower can temporarily be suspended (saving the battery). In my bridge analogy, the road is still present, but the bridge has lifted it out of the way to allow a boat to pass underneath. If the server sends data to the device, the tower sends a radio signal to the device, reestablishing the radio connection, allowing the data connection to complete (the bridge lowers down, allowing cars to traverse the road).

This sounds great—why not leave all of your connections open for future transmissions? Due to the number of connections on a cellular network, orphaned connections are cleaned up by the network after a period of time (typically 5–30 minutes). (I suppose that makes the network a developer who wants to pull out the disused lift bridge to put in a riverside condo.)

Is Your App Working with the RRC State Machine?

The presence of the RRC state machine tells you that your network connections exact a power price that is larger than you may have thought in the past. By grouping connections and making sure that active radio time is minimized, you can greatly improve the performance of your mobile apps. All data connections cost at least 10 s of active use (equivalent to 5 minutes of standby time). It has long been considered that downloading data as quickly as possible is important for performance, but in mobile it is clear that downloading quickly and turning on the mobile radio as infrequently as possible is even more crucial to save resources.

Using Data During a Phone Call

We are in a multitasking world. Talking on the phone while using an app is a very common occurrence. If your app is connected via LTE, and a phone call comes in, the phone will drop to a 3G data connectivity for the remainder of your data session.

This is due to circuit switched fallback. Until Voice over LTE (VoLTE) becomes the norm, all voice calls transmit on the 3G circuit switched network. This means that any active data session will also drop to 3G. In "Battery Historian 2.0" on page 62, we looked at a study in which I was streaming a teleconference while listening on the phone. Looking closely at the time of the phone call, we can see that the radio connection starts blue (LTE), but goes black (HSPA) for the duration of the phone call (Figure 7-2). When the call ends, the radio is able to transition back to the LTE (blue) data network.

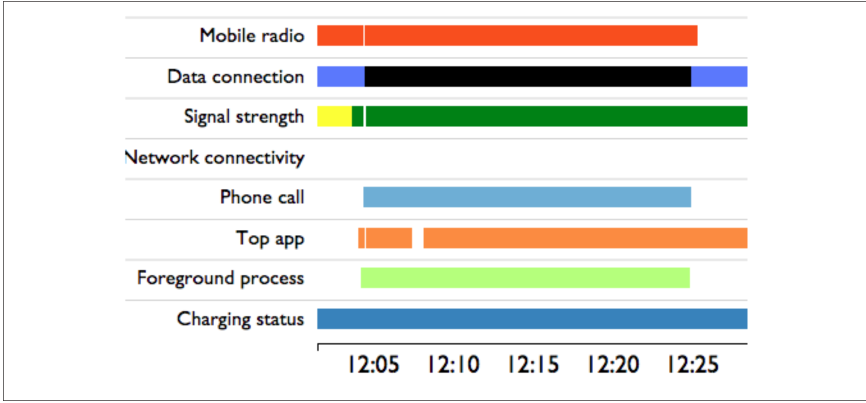


Figure 7-2. Battery Historian showing circuit switched fallback

Testing Tools

So far, we have discussed the power usage of Android's radios, and how mobile data networks function. How do we use this knowledge to optimize our Android app traffic? And if we have optimized our traffic, how can we test it to make sure? There are a number of tools that capture mobile data traffic and allow you to analyze the data. For years, tools like Wireshark and Fiddler have been used by network ops professionals around the world to collect packet data and analyze it for potential issues/optimizations. Man-in-the-middle (MITM) tools help you to decrypt HTTPS traffic to understand what data you are sending securely on the network. These tools are certainly on the front line of mobile app performance. A similar tool called the AT&T Application Resource Optimizer (ARO) records packet captures, and additionally

provides suggestions to app developers on how to optimize traffic for cellular connections.

Wireshark

Wireshark is probably the most popular network analysis tool in the world. It is a free tool that runs on your desktop, and it collects packet data traveling across a data connection. Data can be observed in real time, and after the data is collected, it can be saved into a file for further analysis.

To test your Android phone with Wireshark, you must connect your phone to the PC that has Wireshark installed. For Windows machines, I use Connectify to convert my laptop into a Wi-Fi hotspot. When my Android device is connected to the hotspot, all of the data traffic from my phone is going through the computer (and visible to Wireshark). In the Wireshark app, you can now begin collecting on the wireless interface (if you are not sure which one is the Wi-Fi interface, initiate network traffic on your phone, and you'll see one of the interfaces begin to send and receive packet traffic).

As you can see in Figure 7-3, Wireshark shows every packet that is sent back and forth from my phone (*192.168.223.104*) and the computer (*192.168.223.1*). It was initially challenging to make heads or tails of what was going on, so I added a filter for HTTP. This restricts the packets to just HTTP packets, and now I can begin to see the requests and responses that occurred during my testing. You can now tell that I was opening *cnn.com* in the browser, and the requests that follow. Packet 1481 shows that my *cnn.com* request caused a 301 redirect (in the red box) to *www.cnn.com*, and that page spawned a number of requests to turner CDNs for files. If I wanted to discover how many 301 redirects there were in this packet capture, the filter "http.response.code == 301" drills down to show three such redirects.

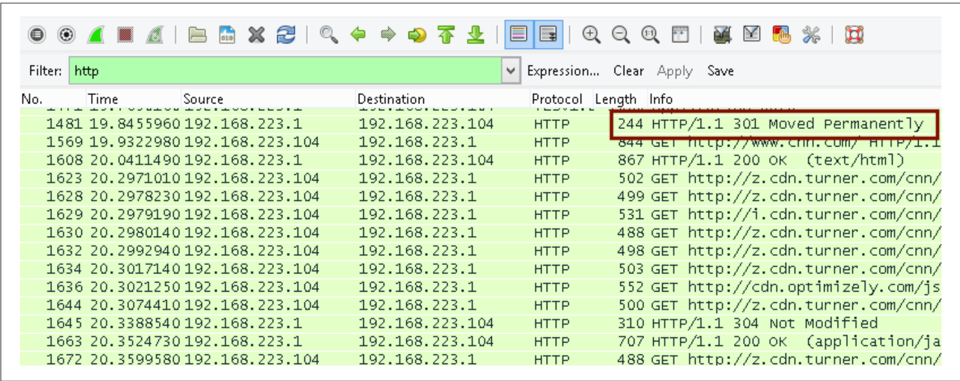


Figure 7-3. Wireshark Packet Capture

The filtering tools in Wireshark are extremely powerful. It is possible to search and filter for specific files, specific types of files, files with cache headers, and more (the

possibilities are endless!). In practice, these are all great searches, but you have to have an inclination to what the issue is—otherwise, you'll feel like you're looking for a needle in a haystack with this approach. However, if you think you have an idea about what your issue is, Wireshark is a great way to drill down and pinpoint it exactly.

Fiddler

Fiddler is another free tool that analyzes network traffic. Fiddler acts as a proxy for all of the data that comes through the device. By acting as a proxy, it can also act as a man in the middle (MITM), allowing you to decrypt HTTPS traffic. In Wireshark, you can see that HTTPS traffic is being transferred, but you are unable to decode it to see what the files are, or what they contain.

Running Fiddler is similar to running Wireshark. I connect my Android device to the Connectify Wi-Fi hotspot. Then, by modifying the Wi-Fi settings on my device to add the Fiddler proxy, and installing the Fiddler certificates on my device and PC, Fiddler can read all of the data coming through the connection (there are excellent instructions at the Fiddler website to complete this setup).

Once this is all connected, you will start to see traffic move through the Fiddler window. In the screenshot shown in Figure 7-4, I was using the YellowPages app to find grocery stores nearby.

#	Result	Protocol	Host	URL	Body	Caching	Content-Type	Process
1…	200	HTTPS	r.ypcdn.com	/2/p/ypmandroid?ptid=yp…	43	no-cache	image/gif	
1…	200	HTTPS	r.ypcdn.com	/2/n/ypmandroid?ptid=yp…	43	no-cache	image/gif	
1…	200	HTTPS	syndication.yellowp…	/v1/business_listings/465…	835	no-cac…	application/json; chars…	
1…	200	HTTP	yellowpages.112.2…	/b/ss/yellowpagesprodyp…	1	no-cac…	text/html	
1…	200	HTTP	Tunnel to	syndication.yellowpages.c…	0			
1…	200	HTTP	Tunnel to	syndication.yellowpages.c…	0			
1…	200	HTTP	Tunnel to	syndication.yellowpages.c…	0			
1…	200	HTTP	Tunnel to	syndication.yellowpages.c…	0			
1…	200	HTTP	Tunnel to	r.ypcdn.com:443	0			
1…	200	HTTP	Tunnel to	syndication.yellowpages.c…	0			
1…	200	HTTPS	r.ypcdn.com	/2/i/ypmandroid?ptid=ypa…	43	no-cache	image/gif	
1…	200	HTTPS	syndication.yellowp…	/v2/my_book/featured_co…	3,304	no-cac…	application/json; chars…	
1…	201	HTTPS	syndication.yellowp…	/shorturls?api_key=b1b92…	1	no-cache	text/html; charset=utf-8	
1…	200	HTTPS	syndication.yellowp…	/v2/directions?api_key=b…	668	no-cac…	application/json; chars…	
1…	200	HTTPS	syndication.yellowp…	/v1/businesses/46541718…	323	no-cac…	application/json; chars…	
1…	200	HTTPS	syndication.yellowp…	/v2/consumer/search?api…	2,513	no-cac…	application/json; chars…	
1…	200	HTTP	i3.ypcdn.com	/blob/c3da396ea1652450…	50,696	public,…	image/jpeg	

Figure 7-4. Fiddler proxy capture

In Figure 7-4, you can see in the left window of the Fiddler packet capture, and the boxed field is a response from the YP app. The file is 668 bytes (on the *wire*), has a cache setting of "no-cache" and is a JSON file. It contains the directions from my house to the grocery store, and the file was encrypted using HTTPS (and this is good, as the response has my address/location in the file). On the right side of the Fiddler window (seen in Figure 7-5), there are a number of windows with lots of options. The

top window is showing the headers sent to *syndication.yellowpages.com*, and the bottom window shows the decrypted response. Inside the JSON file, you can see that the total distance to the grocery store is 4.787665 miles (that's some pretty serious accuracy!). The application further predicts the drive time to this store as 661 s, or just over 11 minutes away.

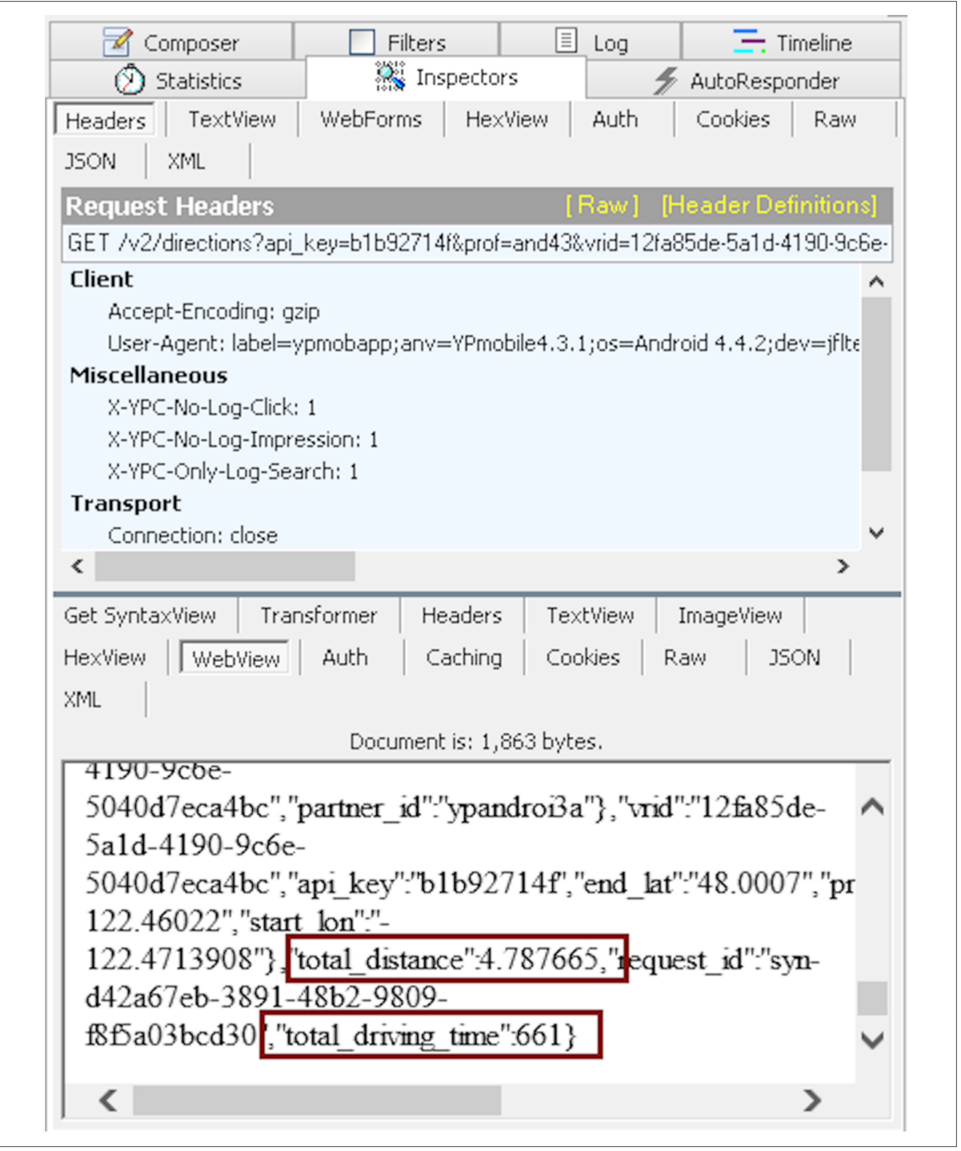


Figure 7-5. Fiddler proxy capture detail windows

MITMProxy

MITMproxy is a tool similar to Fiddler that creates a MITM and allows you to decode HTTPS traffic going across your network.

The ability to decrypt HTTPS traffic is an incredibly useful tool, as much of your data traffic uses HTTPS to secure your customers' data. By decrypting the data, Fiddler and MITMproxy also allow you to ensure that the correct files and information are being sent to any third-party SDKs that you have added to your app.

AT&T Application Resource Optimizer

The ARO is a tool specifically designed for monitoring network performance in Android and iOS apps. It is a free/open source tool from AT&T, and it contains much of the same packet capture functionality as Wireshark and Fiddler. Unlike Wireshark and Fiddler (which require Wi-Fi connections to a computer), ARO has the ability to collect packet traces over the cellular network. Once ARO has collected the data from your test, it processes it and provides developer-friendly tables and graphs to better help dissect the data. The traffic is graded against 25 mobile networking best practices, giving you immediate feedback on areas for performance improvement. The summary of these tests is shown in Figure 7-6 (the red x indicates a failure, and the green check indicates that the trace passed the test criteria). We'll discuss these best practices throughout this chapter.

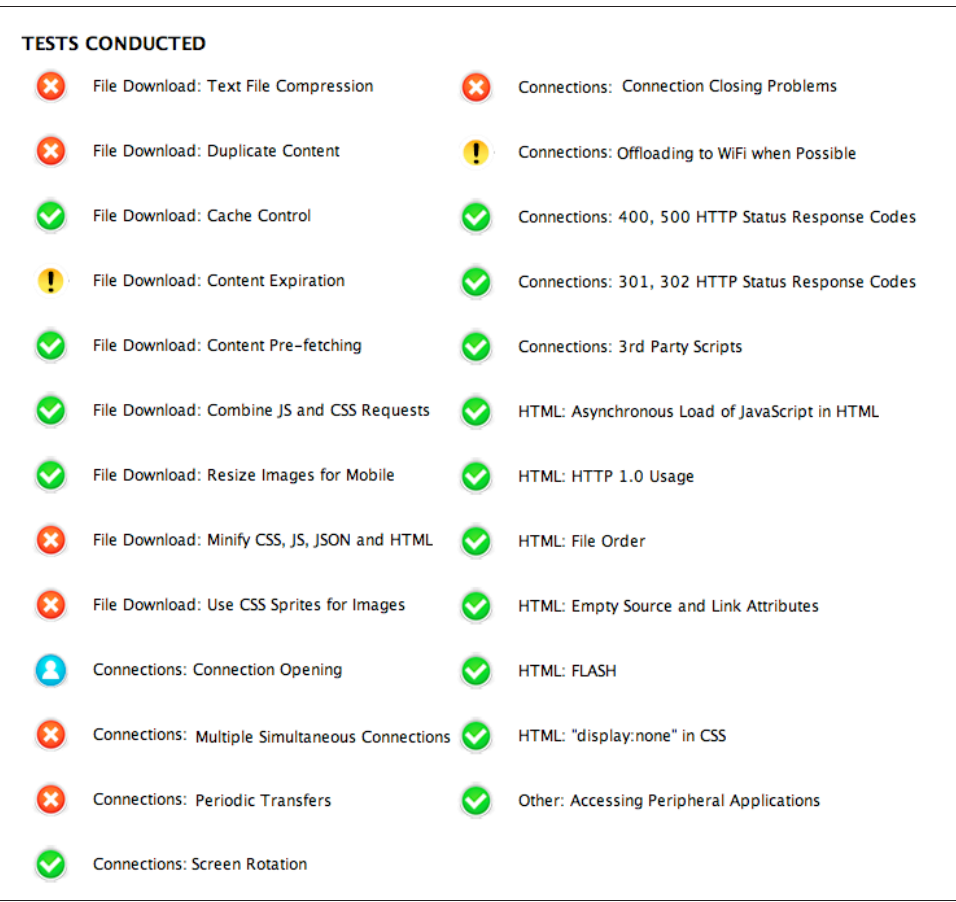


Figure 7-6. ARO best practices: pass fail

There are two versions of ARO for Android devices. The ARO Data Collector APK runs a TCPdump collection directly on your device, gathering all the packets (and assigning each connection to a process). This requires a rooted Android device, and so to simplify testing, a version that does not require rooting is also available. Without root, we lose the ability to assign connections to specific processes, which makes it a bit harder to pin traffic to a specific app (if there are a lot of apps running on the device).

Once you collect a data trace in ARO, you can analyze the trace in the ARO Analyzer tool. The test you performed in your app will be graded against the 25 best practices shown in Figure 7-6. Each best practice is further enumerated with additional details, so if you fail any of the best practices, you can learn why and how you failed (Figure 7-7).

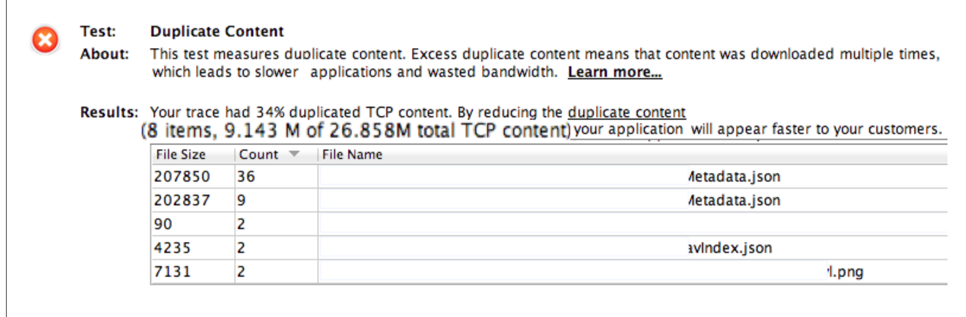


Figure 7-7. ARO duplicate content best practice; 34% (9 MB) of data sent multiple times is a lot! (Some text enlarged for readability)

There are five tabs of data provided out of each trace, but the Diagnostics tab is where you can really see how the data flows in and out of your app (Figure 7-8).

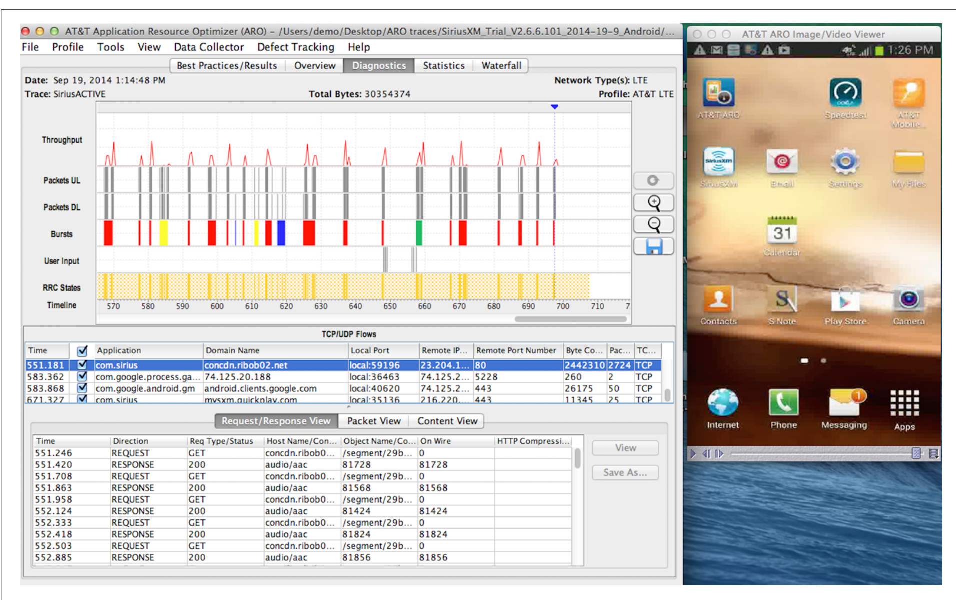


Figure 7-8. ARO Diagnostics Tab

The Diagnostics tab contains a lot of information, so let's look at all of the data presented here. There are two windows shown: on the left is the data analysis, and on the right is a video of the screen taken during the trace. The video is synced with the trace, so when selecting a packet or a connection, you can see what was on the screen at that given moment.

Looking more closely at the Diagnostics tab graph in Figure 7-9, the chart graphs the packet traffic over time. The top row shows a normalized throughput over time. This

allows us to see traffic that uses a relatively large amount of data, versus connections that use smaller amounts. The next two rows show the packets being uploaded and downloaded over the connection. The row labeled "Bursts" describes the typed of traffic based on the color. Red bursts are initiated by the app, yellow by the server, green occur after a user input event (recorded on the next row), and the blue bursts show mostly empty packets that are the result of connections being closed. The bottom row shows the LTE RRC state machine, which is outlined in Figure 7-1. Solid color indicates continuous reception, and the cross-hatched region represents the Tail. Blank areas show when the radio is off (Idle). Note the blue arrow and dotted blue line. This signifies the moment in the trace displayed on the video viewer. If you were to press play on the video, you would see the line move to the right, allowing you to see the packets being transferred while also watching what was on the device screen.

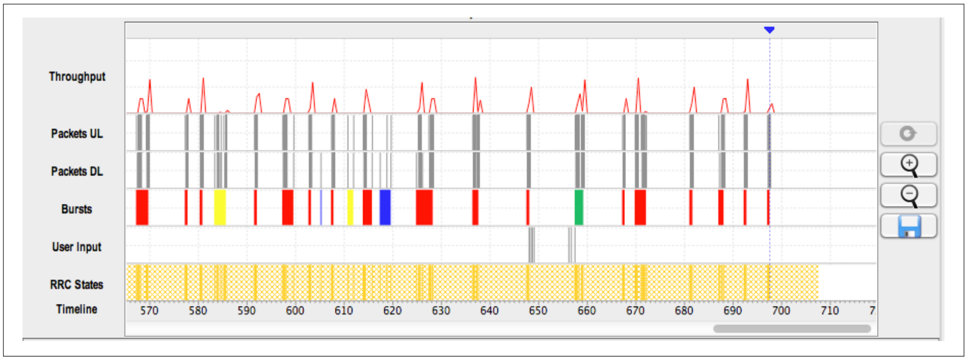


Figure 7-9. ARO Diagnostics tab graph

The two tables below the graph provide more insights into each data connection that occurs in the network trace (Figure 7-10). The top graph shows every TCP or UDP connection that was initiated during the trace. Currently highlighted is a connection started at 551 s from the SiriusXM Radio app. You can see the domain and IP information, as well as the byte count and packet count for the connection.

The bottom table shows the requests and responses from the highlighted TCP connection in the top table. This table shows us that there are a number of 81 KB music files transferred in this connection.

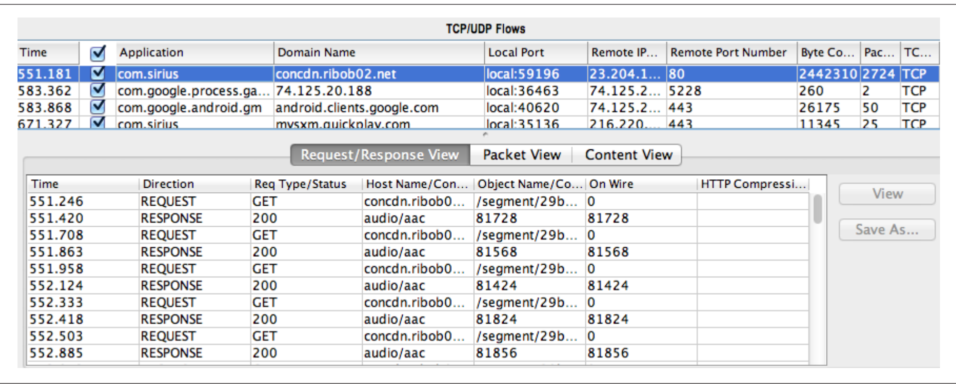


Figure 7-10. ARO Diagnostics tab tables

There are several additional views in ARO that provide an extraordinary amount of information, and we'll see additional screenshots through this chapter as we discuss potential optimizations.

One disadvantage to ARO is that it cannot parse any details from files sent via HTTPS. If your app uses HTTPS, you'll need to manually look at those files in a Fiddler trace.

Hybrid Apps and WebPageTest.org

WebPageTest.org is a great tool for testing websites. I know, you're thinking "This book is on Android app development, so why am I talking about website tests?" Thousands of Android apps are built with tools like PhoneGap, that simply wrap components from websites with native code, allowing a more native app experience. This native app generally just displays the content in an embedded web view to appear as if it were native. Because the wrapper is not possible to optimize, as a hybrid app developer, you can only work to ensure that your web components run as quickly as possible.

WebPageTest allows you to test your website from several locations around the world, but the Dulles, VA, location has several Motorola and Nexus devices available for testing (with Chrome and Chrome Beta). The tests in WebPageTest will show you how fast your web page loads on mobile, and indicate areas for improvement.

Network Optimizations for Android

The web performance community has established a number of best practices for websites, and these also apply to mobile apps. We will also discuss several additional *mobile-only* best practices for your Android app (and they also apply to any iOS development you might do). The optimization best practices listed here are in no par-

ticular order of importance, as each of these will affect app performance differently (and some will likely not apply to your app at all). The general trend for network performance (whether on desktops or mobile) is to download everything as quickly as possible. By getting out of the way of the radio, and letting the radio turn off, you save power. By getting the content to your customers as fast as possible, your customers are more engaged and less likely to become frustrated.

The basic rules for mobile app performance basically derive themselves from Steve Souders's iconic list of 14 performance rules from his book *High Performance Web Sites* (O'Reilly):

- Make fewer HTTP requests
- Use a content delivery network
- Add an `Expires` header
- Gzip components
- Put stylesheets at the top
- Put scripts at the bottom
- Avoid CSS expressions
- Make JavaScript and CSS external
- Reduce DNS lookups
- Minify JavaScript
- Avoid redirects
- Remove duplicate scripts
- Configure `ETags`
- Make AJAX cacheable

Now, several of these are website specific, but most of them still hold for Android native optimizations, and we will cover them in the following sections.

File Optimizations

There are two basic ways to download data faster: lower the number of requests (Souders's rule #1), and/or reduce the size of those requests (several of Souders's rules). This can be a hard pill to swallow, as our apps are getting more and more complex each day, but hopefully the pointers in this chapter will help you come up with plans to reduce the amount and size of content in your app.

Text File Compression (Gzip Components)

This is one of the easiest fixes to make. When delivering text files (HTML, CSS, JavaScript, JSON, etc.) to your app, compressing them on the server can reduce the file size by 4–8x. This large reduction in file size from your server to your app means fewer roundtrips and faster delivery of the file. For example, in one app, we saw a 200 KB text file downloaded without Gzip compression. Placing this on a compression-enabled server reduced the size on the wire to 51 KB. Not only do you deliver the content to your customers faster, but you reduce the utilization and bandwidth of your servers too!

There are a number of Gzip algorithms available for use today. In general, the standard Gzip compression is good enough for most apps. If you are really trying to get the most out of compression, and you have files that do not change very often, you could try the Zopfli compression algorithm, as it squeaks out about 5% more compression than the default Gzip algorithms. On the downside, it takes about 100x longer to perform the file compression (hence the "only use it on precompressed files" warning).

Enabling Gzip compression is a simple server change—no code change in your app—you simply need to add the file extensions/MIME types and so on to your *.htaccess* file:

```
<ifModule mod_gzip.c>
mod_gzip_on Yes
mod_gzip_dechunk Yes
mod_gzip_item_include file .(html?|txt|css|js|php|pl)$
mod_gzip_item_include handler ^cgi-script$
mod_gzip_item_include mime ^text/.*
mod_gzip_item_include mime ^application/x-javascript.*
mod_gzip_item_exclude mime ^image/.*
mod_gzip_item_exclude rspheader ^Content-Encoding:.*gzip.*
</ifModule>
```

You'll immediately see your text files downloading more quickly. One additional addition to Gzip might be to exclude small files. Files under 850 bytes will fit into a single packet uncompressed anyway, so while the Gzip compression/decompression has very little overhead, you could still omit these steps for such small files.

ARO tests all text file captured in the trace for Gzip compression. You can discover whether files are Gzipped in two places in ARO:

Best Practice: Text File Compression
: Any text file sent without compression is listed in Text File Compression (Figure 7-11). Currently, there is no way to directly know the savings for adding compression, but the table does report the uncompressed file size. Note that files under 850 bytes are not flagged (because they fit into one packet, and will only require one roundtrip, even without compression).

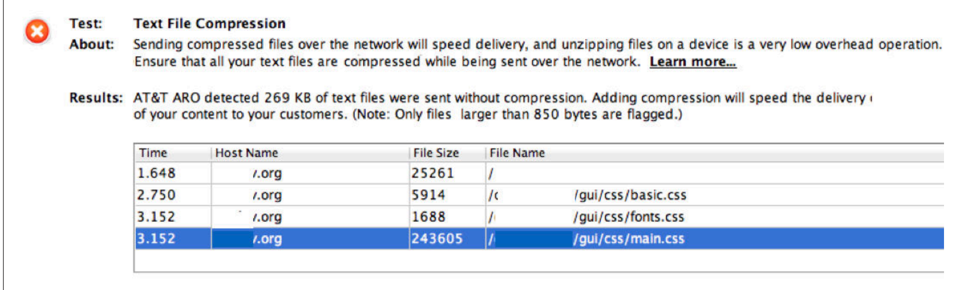


Figure 7-11. ARO Text Compression Best Practice

Diagnostics tab (Request Response table)

In Figure 7-8, the bottom table shows the requests and responses. The rightmost column will identify text files with compression, or have "none" for files with no compression.

Text File Minification (Souders: Minify JavaScript)

Another way to shrink the size of your text files is through the process of minification. Minification is a process that takes out all of the human-readable formatting to your text files (like whitespace, tabs, and comments) to make the files smaller. For example:

```
<html>
 <title> A Sample Page</title>

<body>
with some sample text
<--do more here-->
</body>
</html>
```

becomes:

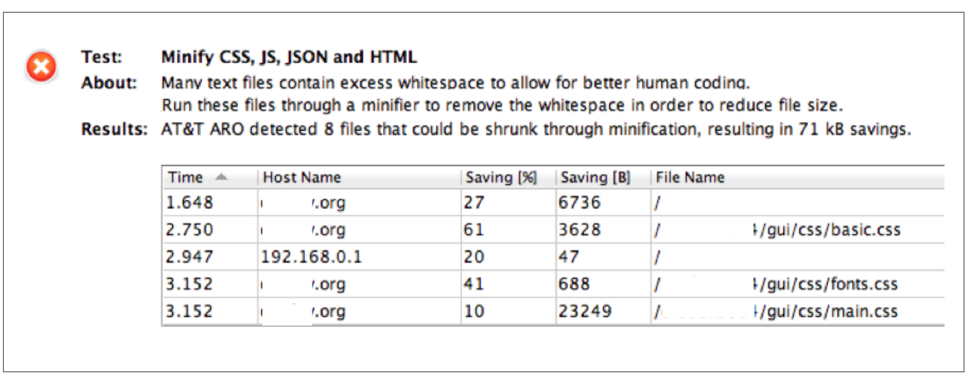
```
<html><title> A Sample Page</title><body>with some sample text</body></html>
```

Depending on the size and complexity of your page, minification can save up to 20%–50% of the file size. Many build tools (like grunt) include Minification libraries that can automatically minify your files whenever you make changes (saving you work too!).

One might argue that using Gzip is enough to reduce the transmission costs of a text file. For example, minification might reduce the file size by 10%–15%, but the difference in Gzip savings for the minified versus not-minified file might only be 1%–2% (because whitespace compresses well).

However, you should always minify before Gzipping, not to save the 1%–2% of network transmission, but because it will offer storage savings on your customers' devices. In addition, reading a smaller file into memory is faster (and less likely to induce a crash on devices with limited memory).

While the Souders rule looks at only JavaScript for minification, this optimization can be run on any text file to reduce the file size. The ARO tool looks at all CSS, JavaScript, JSON, and HTML files for minification opportunities (Figure 7-12). It calculates the potential savings on each file, and provides a total savings for the files captured in the trace.



Test: **Minify CSS, JS, JSON and HTML**
About: Many text files contain excess whitespace to allow for better human coding.
Run these files through a minifier to remove the whitespace in order to reduce file size.
Results: AT&T ARO detected 8 files that could be shrunk through minification, resulting in 71 kB savings.

Time	Host Name	Saving [%]	Saving [B]	File Name
1.648	.org	27	6736	/
2.750	.org	61	3628	/ /gui/css/basic.css
2.947	192.168.0.1	20	47	/
3.152	.org	41	688	/ /gui/css/fonts.css
3.152	.org	10	23249	/ /gui/css/main.css

Figure 7-12. ARO Minification Best Practice

Images

When it comes to apps, images are the most commonly downloaded file type. They are also among the largest files in size, and because images are everywhere, easily recycled from your web page or other digital service. Controlling image size is a fantastic way to reduce the data usage in your mobile app. There is a balance between image quality and image size that you must discover for your app (probably with the help of UX and editorial teams). Once you find that correct balance, you'll have a great-looking app with images that are optimized to download and render quickly.

Super Size It?

If you create one version of every image in your app, and serve it to every mobile device (including the retina display tablets from that *other* mobile OS), you will likely want to use an image that looks great on all of those devices (meaning that the image downloaded will probably be pretty big). Now, imagine how long it will take to deliver an image sized for a Retina-enabled tablet to a small Android devices on a 2G network. This is probably not the user interaction you are looking for.

To account for all of the various screens sizes, you may want to create image buckets for your images, and whenever an image is needed, your app can provide the screen

size to ensure that the correctly sized version of the image is delivered. Android has provided screen resolution buckets for app development, and these values are a good start for images on the network too.

If your app uses thumbnail images as well as full sized, you may want to consider delivering thumbnail sized images for images where the full-sized image might not be needed.

Dumbnails

One popular app I worked with was using 250 KB images for the thumbnails next to articles. With six to eight article titles on the page, these tiny images added 1.5–2 MB of data on every startup of the app. The developer colorfully described these as "dumbnails" due to their size, and by the next release had small 5–10 KB images in place.

The actual sizes you may choose are very much app independent. You know from the layouts how large the images must be on the screen, so work out popular pixel sizes for the images based on small, medium phablet, and tablet screen sizes. From there, you can work out the correct image buckets for your app.

Metadata

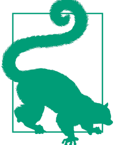
When you take a photo with a digital camera, it is likely that there is metadata connected to the file (this can include information about the device, the settings on the device, and more recently, the location the photo was taken). Photo editing software may add additional metadata to the image. Unless your app is a photography app that discusses the way each photo was taken and how it was edited, you can strip out all of the image metadata to save anywhere from a few bytes to tens of kilobytes with no loss of image quality to your customers!

Compression

Just like with text files, you can compress images to make them smaller, take up less space on the device, and also take less time to download. Image compression is too large a subject to cover in detail here, but at a high level, when you compress images, you tend to lose image quality (*lossy* compression).

The amount of lossy compression applied to your images might depend on the use. For thumbnails, perhaps a higher compression is possible, as they tend to be small, and the "graininess" or pixelation is harder to see. For images inside an article, perhaps saving the JPEG at 70% compression will suffice. For apps focused on photography and graphics, you may decide to not do any lossy compression. The amount of compression is a delicate balance of optimizing the appearance of graphics versus size

compression for speed. Google's PageSpeed server uses image compression of 85% by default, so this might be a good starting point for image comparisons.



WebP: A Successor to JPEG?

WebP is an image format that is being developed by Google. It is generally 20% smaller than a similar JPEG. Support for WebP is growing across browsers and devices (and is supported in Android 4.0 and newer). WebP image format might be worth considering for reducing your image file size.

File Caching

If there are files that are used frequently in your app, you should download these files *once* and store the file locally for reuse. When it comes to performance, reading a file locally will always be faster than establishing a connection and downloading the file. For this performance reason alone, caching will speed up the rendering of your Android app. By avoiding network connections, you are saving capacity on your server, and you are reducing the battery drain of your customers.

Of course, the primary reason to invoke caching is that mobile data plans are constrained by data usage, and downloading excess content could end up costing your customers money if they exceed their monthly cap of data. There are two dimensions to caching: first, you must turn on caching in your app on the device, but then you must also properly set cache times on the server.

Caching in your app

Interestingly, caching is off by default in Android, and you must turn it on. For Android 4.0 and higher, you enable the HTTP response cache by invoking in your `onCreate`:

```
private void enableHttpResponseCache() {
  try {
    long httpCacheSize = 10 * 1024 * 1024; // 10 MiB
    File httpCacheDir = new File(getCacheDir(), "http");
    Class.forName("android.net.http.HttpResponseCache")
         .getMethod("install", File.class, long.class)
         .invoke(null, httpCacheDir, httpCacheSize);
  } catch (Exception httpResponseCacheNotAvailable) {
    Log.d(TAG, "HTTP response cache is unavailable.");
  }
}
```

And now your app will cache!

Caching on the Server

The cache time for each file saved on the device is set in the headers when delivered from the server. When setting up your caching parameters on the server, there are several important considerations that must be taken into account. Typically files are set to cache for a set amount of time, and if the file is requested again during that time period, it is served from the devices' cache. If the time has expired, a connection is made to the server to check if the file has changed. If the file has not changed, a HTTP 304 "not modified" response is sent to the device, and the cache timer reset. If the file is different, the new file is downloaded.

The length of the cache timer really depends on the content, and how often it changes (e.g., sports team logos that rarely change can be cached for a year, weather conditions for 5 minutes, and headline feeds might never cache). By modifying the cache time for your content, you ensure the data your customers see is always *fresh,* but also limit the number of files downloaded in a duplicate manner, saving battery and data. In general, there are three headers you can use to supply the expiration date of your content.

Cache Control (Add an Expires Header)

The header most frequently used for caching is the `Cache-Control` header, which has a few common values that you can assign:

Private/Public
: This is typically used by CDN caches in the network. It tells the CDN if the files are public (can be used by anyone), or if they are private files just for the user.

no-store
: If your files use this term, the files cannot be cached, and thus must be downloaded every time.

no-cache
: The no-cache header is a bit misleading in its name. A file with a no-cache header can actually be cached, but it must be revalidated before reuse.

max age=X
: The max-age denotes the amount of time (in seconds) that a file might be cached. Common values are 0 (same as `no-cache`); 60, 300, 600, 3,600 (1 hour), 86,400 (1 day), 3,153,600 (1 year).

ETags

The `ETag` is a response header with a unique string of random characters. Every time the file is to be used from the cache, the `ETag` must first be validated at the server. If the local string matches the server, the server replies with a "304 not modified," and

the local file is used. If the ETags differ, the new file is downloaded and stored in the cache. Its behavior is the same as Cache-Control: no-cache, or max-age=0.

For files that regularly expire, ETags are a great way to validate that the locally cached file is still in sync with the server. For files that rarely change, ETags are an expensive (from a performance view) caching mechanism. While the file is not downloaded (thus saving bandwidth), a connection is still established, adding connection time to the file processing.

In the following example, both an ETag and a Cache-Control header are present. The device will read from the cache for 86,400 s (1 day) and after that, will check the ETag (or the last-modified) headers to see if the file has changed. If it has not, it will used the cached file for another 86,400 s:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: max-age=86400
Content-Type: image/jpeg
Date: Tue, 28 Jan 2014 00:14:55 GMT
Etag: "b17ad00-1f17-46723595372c0"
Expires: Wed, 29 Jan 2014 00:14:55 GMT
Last-Modified: Thu, 09 Apr 2009 18:23:47 GMT
Server: Apache/2.2.3 (CentOS)
X-Cache: HIT
Content-Length: 7959
```

Expires

Less common than Cache-Control or ETag (but just as valid) is the Expires header. Rather than giving the time in seconds that the file expires, it gives an exact date in the future when the file will expire and should be revalidated. This was original cache header used in the web, and some ancient browsers may still use it. The Expires header should match the Cache-Control: max-age. In the preceding example, this is the case: the Expires header is exactly one day after the file is served.

Faster Than Caching?

Do you download content on the first startup of your app, and then cache them for a long time? Remember that your initial startup is almost your *make or break* point for customer satisfaction. If the first time your app starts up, it takes a long time to configure (as you are downloading images and files), your customers might stop using your app after one visit. Consider placing these images and icons into the resources file of your app. Sure, it makes the app download a bit larger, but this one time download cost will speed up that first startup. And, if you change the logos, icons, etc., all you need to do is issue an update to the app.

To discover if your app is correctly caching, you can use ARO. There are three best practices that help you determine any issues with caching files: duplicate content, cache control, and content expiration. The table in Figure 7-7 shows a list of files downloaded more than once in a trace, the number of times each file was downloaded, and the size of the duplicated files. The Cache Control and Content Expiration Best Practices in ARO serve as warnings for potential caching issues on the server or on the device (respectively).

The ARO Cache Control best practice is looking for the presence of a `Cache-Control/ETag` or `Expires` header. If no such header is inserted by the server, it throws a warning: "this is where your caching policy might be failing!" There are valid times for this to fail. Perhaps you have a file that you don't want to cache, so you leave out the headers. It is important to note that the HTTP cache spec says if the file says nothing, it can be cached for 24 hours. If you do not want the file cached, make sure you say so explicitly to avoid any future literal reading of the spec!

ARO's Content Expiration best practice is looking to make sure that your app's cache is working correctly. It counts the number of 304 not modified server checks, as well as the number of times the cache header is ignored, and the file is requested from the server (as the file should be in the cache). Typically this flashes a warning on apps whose cache is not configured (or configured correctly) on the device.

If your app is downloading content in a duplicate manner, take a close look to ensure that the headers are populated properly (server fix), and that your app is storing the files correctly in its cache (application fix).

Beyond Files

Optimizing the files that you download is crucial. Smaller, leaner files will always lead to a faster download, and zippier performance. However, now you also know about how cellular latency and "RRC State Machine" on page 176 can affect download speed and battery drain. Assuming all of your files are now optimized, let's make sure that the processes you use to connect your app and server to get these files are running as efficiently as possible, working with the state machine to maximize performance and customer satisfaction.

Grouping Connections

Imagine an ad-supported image sharing app. Intuitively, we know that connections serving images will consume a lot of bandwidth, and connect to the network frequently. Do you know how the ad SDK will behave? Do the ads load *with* your images, or do these connections occur when the radio would be otherwise silent? How about your analytics data? Do these connections fire at the same time as your app? Or do they wake up the radio whenever they want to?

As you might imagine, many of these tools are built to *just connect*. If you use more than one analytics provider (or ad service) in addition to other libraries and connections, your app might never let the radio go to sleep, due to all of the services connecting whenever they want to. If you are looking to ensure that your app is as efficient as possible, it makes sense to organize your connections into as few large buckets as possible (versus many small buckets). Look into the documentation and code for these SDKs, and see if you might be able to sync them with other connections from your app. If you test, and see that your third-party SDKs are not behaving as nicely as they should be, reach out to the developers. Odds are they too are unaware of the way their libraries behave, and would be interested to improve their libraries' network behavior.

Regular Connections

Because every connection to your server keeps the radio on due to the RRC state machine, it is crucial to minimize the connections that occur in your app (especially those that occur in the background) to not only preserve the battery life, but also the data plan of your customers. In 2013, my team worked with a popular social media app to reduce the number of connections that the Android app made in the background. In this early version of the app, we saw three connections running in an uncoordinated way in the background. Every 30 minutes, these three connections turned on the radio seven times. In Figure 7-13, 30 minutes are bordered by the packets with red "Bursts." In the top example, you can see two closely occurring (but not overlapping) purple, yellow, and blue bursts. The purple connections open the second and third connection, the yellow bursts are reusing these two connections, and then blue is a packet from the server telling the app to close the connection.

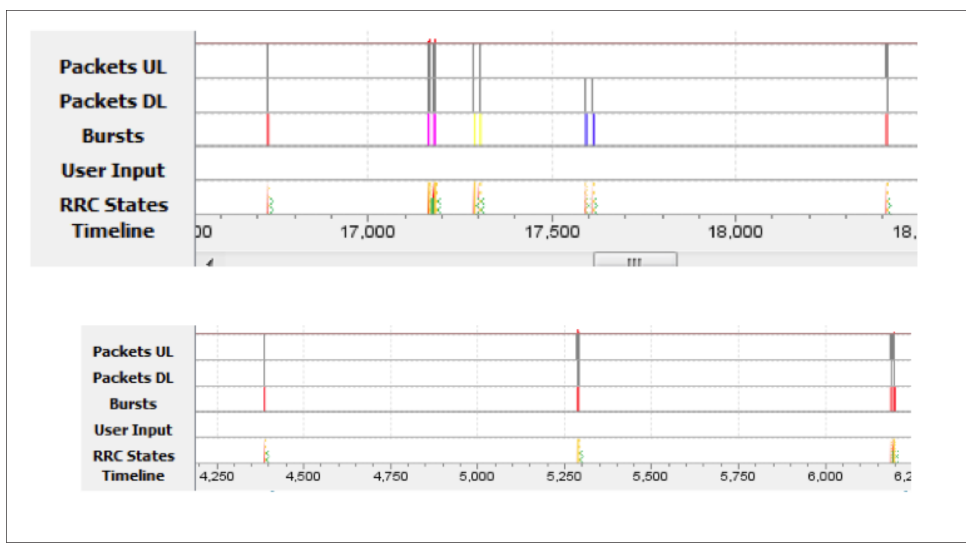


Figure 7-13. Social media connections in the background: before (top) and after (bottom) optimization

When the developers saw this, they realized that by simply coordinating these connections (and making sure to close them properly), they could greatly reduce the background battery drain of their app. The bottom trace shows the improvement. The "updated" version of the app pushed all three connections into one transaction time, and the developers doubled the refresh rate to every 15 minutes. Now, with two connections every 30 minutes, the data is being updated twice as often, but the battery drain actually decreased by >50%. Assuming that these connections occurred 24 hours a day, we estimate that this actually saved ~5% of battery usage for every customer with this app installed!

Not all developers have the time to build their own transaction managers, but the Android developers have been listening. In "JobScheduler" on page 66, we looked at the JobScheduler API, which was introduced in Lollipop, and the examples showed how letting the OS handle periodic connections reduced what could have been 20 connections to a mere 9! By adding the flexibility of the JobScheduler API to download non-crucial elements in a more flexible manner, and by placing a fallback mechanism on background connections, your radio usage will decrease dramatically, improving the performance of your app while simultaneously using less battery (a win-win for you and your customers)!

Detecting Radio Usage in Your App

To determine if your customer's device is connected to Wi-Fi or cellular, you can query the connectivity manager, as shown in Example 7-1.

Example 7-1. Identify Connection: Wi-Fi or Cellular

```
public static String getNetworkClass(Context context) {
    ConnectivityManager cm = (ConnectivityManager)
                             context.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo info = cm.getActiveNetworkInfo();
    if(info==null || !info.isConnected())
        return "-"; //not connected
    if(info.getType() == ConnectivityManager.TYPE_WIFI)
        return “wifi";
    if(info.getType() == ConnectivityManager.TYPE_MOBILE){
         return “cellular";
         }
    }
    return “unknown";
}
```

With this snippet of data, you now know what sort of connection is in use, and you can customize the data stream for the two network types. If your users are on cellular, you might have non-urgent communications that you can delay transmission. Prior to the Job Scheduler in Android Lollipop, there was no way to tell if the network was being used. To force analytics and ads to only load when the cellular radio was already in use, I used the code shown in Example 7-2.

Example 7-2. Identifying the presence of a cellular connection

```
 if (Tel.getDataActivity() >0){

          if (Tel.getDataActivity() <4){

          //1, 2, 3 response means that the cellular radio is transmitting!
          //download the image here using image getter
           imagegetter(counter, numberofimages);

           //and show the ad
           AdRequest adRequest = new AdRequest();
           adRequest.addTestDevice(AdRequest.TEST_EMULATOR);
           adView.loadAd(adRequest);
           // Initiate a generic request to load it with an ad
           adView.loadAd(new AdRequest());
  }}
```

This code snippet uses the `TelephonyManager (Tel)` data activity APIs to determine if the radio is on, and if it is on, piggybacks on the connection to download more content. This will only indicate if data transmission is occurring on the cellular network (not on Wi-Fi). In Lollipop, new APIs were added to `ConnectivityManager` abstracts this method from just cellular to all radio connections with `ConnectivityManager.OnNetworkActiveListener` to find out when the radio is in a high-powered state (and ready to transmit data). To see if a network is already active, you can use `ConnectivityManager.isDefaultNetworkActive()`. Using a radio connection that is already established is a great way to share the resources, and save customer battery.

GCM Network Manager

At Google I/O 2015, Google and Android made scheduling battery efficient network connections even easier. As a part of Google Play Services, they added GCM Network Manager APIs that mimic the JobSchedule API for connectivity. However, JobScheduler only runs on devices using Lollipop and newer, while the GCM Network Manager runs on all Google Android devices back to Gingerbread (2.3)! Now, just like in JobScheduler, you can easily set your connections to only run when on Wi-Fi, or when the device is plugged in. You can set tasks to run periodically in the background, or to automatically back off. By utilizing this API for your non-urgent updates and connections, you will directly save a large amount of device battery for your customers.

All Good Things Must Come to An End: Closing Connections

With the latency to establish radio and TCP connections on cellular, you might think it sensible to just keep a TCP connections to your server open. That way, if more packets need to be sent to the device, you can reduce some of the latency on connection setup. This is the case for files sent in relatively rapid succession. But if the files are separated by 15 s or more, the radio will likely still have to be turned on, and you'll save *very* little time on the connection setup.

If connections are left open with no data traffic for a period of time, either the device or the server closes the connection as a cleanup process. This is also not a bad thing (as you'll see in the next section). However, what is negative about this is that the side closing the connection will tell the other party "Hey, I am closing this connection now" and this can lead to the radio turning on, and running through the 10–15 s RRC state machine on the device, causing extra battery drain for your customers.

In the screenshot from ARO shown in Figure 7-14, a small image is downloaded at 8 s, but the connection is not closed.

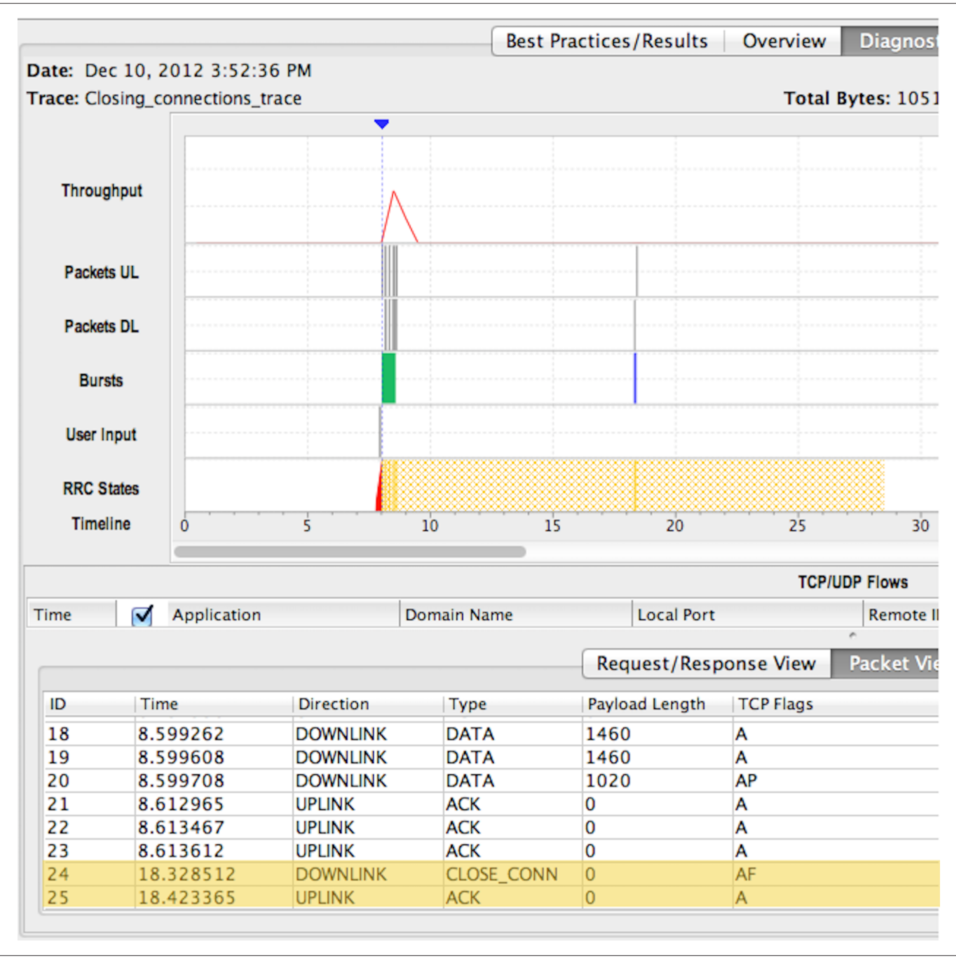


Figure 7-14. ARO Diagnostics tab showing connection closing issue

At 18 s, the server (likely doing a cleanup process) closes the connection, causing the RRC timers to reset (in the packet view table: packet ID 24 at 18 s comes from the server to close the connection). Instead of turning off at ~18–19 s, the radio remains on until 28 s—nearly doubling the battery drain for one image.

For connections where you know that you will not be needing the connection any longer, you can specify that the connection should be closed when you are completed with the download. In Example 7-3, I disable the connection keep-alive. Finally, when the connection has finished its download, I disconnect it. This tells Android that the resources for the connection can be reused or closed (saving memory, etc.).

Example 7-3. Properly Closing Connections

```
HttpURLConnection connectionCloseProperly = (HttpURLConnection) ulrn.openConnection();
//this disables "keep-alive"
connectionCloseProperly.setRequestProperty("connection", "close");
connectionCloseProperly.setUseCaches(true);
connectionCloseProperly.connect();
Object response = connectionCloseProperly.getContent();

InputStream isclose = connectionCloseProperly.getInputStream();

        ...download and render bitmap image

connectionCloseProperly.disconnect();
```

When this code is implemented, the image is downloaded, and the server and the device immediately closes the connection.

Regular Repeated Pings

For apps that require data updates at regular intervals, tools like Google Cloud Messenger should be used to push this information down to the app. Building your own service often results in polling in the background by setting an alarm for every *x* minutes, then waking up the radio and downloading your data. This does not seem like a big deal, but imagine an app that pings the server for updates every 3 minutes. Extrapolate this out—your app will make 480 connections every 24 hours. Throw in a 10 second state machine timer, and now these "harmless" connections are using 80 minutes of radio time per day. If you must have a regular wakeup for data, make sure that you have a fallback procedure, or disable the alarm after a certain amount of time to keep your app (and the device) asleep.

There are cases (think real-time games) where the app may need to keep packets going back and forth on a connection. Make sure you are minimizing the data being sent, but also know that keeping the radio connection on while your app is running can cause major battery drain.

A Perfect Storm: Repeated Connections and Closing Connections

Imagine an app that sends real-time data every five seconds between the phone and a server to update the locations of several people as the move around an area. Now, imagine that the connections are left open on the server for 90 s after the last packet is received (reserving the IP address on the server). In general, if each user is using one connection per session, this should not be an issue. But what if you changed a configuration in your code so that each one of these pings opens a new TCP connection to the server, and your testing did not catch this before release?

You have generated a Perfect Storm of data traffic. Now each Android user is pinging every five seconds, using as many as 18 IP addresses to your server. As more users connect, you begin to see IP collisions, and users are unable to connect! Congratulations, your app has just successfully completed a distributed denial-of-service (DDoS) attack against your server.

Be very careful with repeated pinging of the server, and always test before release.

Security in Networking (HTTP versus HTTPS)

When transporting data over the network, it is imperative that you keep your customers' private data secure. It seems that not a week goes by without a serious breach of private information from a mobile app. Properly storing the files locally on the device and on your servers is crucial, but so is transporting those files back and forth across the network. Your customers may connect to any sort of network, including compromised Wi-Fi hotspots in cafes. If the data you transmit is sent via HTTP, the snooper can get that data with no effort at all—you sent it in cleartext! By using HTTPS, you encrypt the data using an encrypted key. Sharing this key can result in an additional roundtrip when the connection is initialized, but assuming that you have correctly configured your HTTPS connection, it is considered secure.

Worldwide Cellular Coverage

Realtors have a mantra about finding the right house: "Location, location, location." If you totally optimize your networking for all of the best practices just described, you have made an excellent start at mobile network performance. However, the most important variable that we have not accounted for is the speed of your customers' networks. We (obviously) cannot control how or where our customers connect to our app. However, we can work to make sure that the experience is as optimized as possible.

According to GSMA Intelligence, in 2014, smartphones account for nearly 40% of all cellular connected devices (and this will grow to 65% by 2020).

In examining Figure 7-15's global market penetration in Q3 2014, we see that ~5% is 4G, and 3G is just a hair above 30%. This implies that there is a sizable audience (at least 5%, if we assume zero 3G feature phones) of smartphones running exclusively on 2G networks.

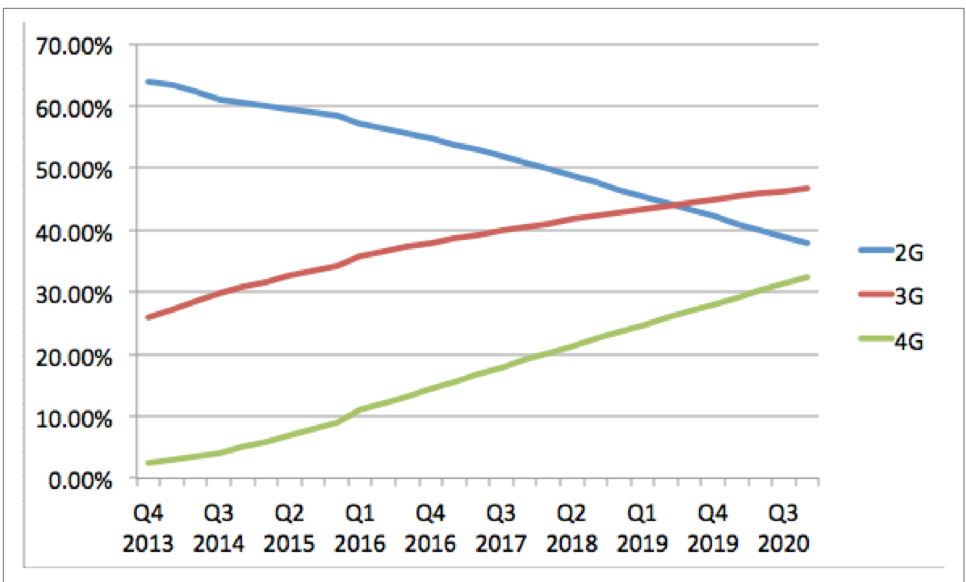


Figure 7-15. Global Market Penetration by "G" (Courtesy GSMA Intelligence)

In 2013, Baidu reported 270M Android users in China, and 31% of those relied on 2G networks for connectivity. Since then, LTE has launched in China, but it seems pretty clear that many Android users are still relying exclusively on 2G for data connectivity.

Smartphones running on slow networks is not a problem limited to areas outside the Unites States and Europe. Even in the developed world, there are places where LTE rollouts have not yet occurred (or high-speed coverage is sporadic). Your customers will travel to those places and try to use your app, so it makes sense to ensure that your mobile app runs well on slower, more congested networks.

In Chapter 2, I discussed how "Your Devices Are Not Your Customers' Devices" on page 14. The same can be said about your mobile network. Most developers live in areas of high network coverage, with at least 3G (and probably 4G LTE) radio connectivity. In addition to the large screens and fast processors, we as developers live in a bubble of highly available networks. For customers in similar areas around the world, this is great. However, it is useful to look at network connectivity around the world to ensure that we are indeed properly serving data to our end users.

CDNs

As latency is a major stumbling block in cellular data communication, anything you can do to reduce latency to your end users will speed the delivery and thus the rendering of your app. While the speed of light is incredibly fast, it still takes 53 ms to make a roundtrip from Boston to London (and Boston to Sydney is 162 ms)! In order

to reduce this latency, consider using a content delivery network (CDN) to mirror your content in data centers around the world allowing your customers faster access to the data they are requesting.

CDNs (at a very high level) are servers that store your data *at the edge* or *near the last mile*. By relying on a distributed system of data stores, your main system is not overwhelmed with requests, and by placing these CDNs near your customers, you get the data closer to them, thus reducing the roundtrip time to request and deliver the files.

In Indonesia, Facebook reports that 50% of mobile users utilize Facebook, yet 75% of customers rely on 2G networks. Again faced with a large customer population with a limited connection, Facebook worked to realize the biggest gains possible. Indonesia is a large archipelago, covering a huge amount of distance. However, most of the country is 2,000 miles from Singapore (a likely CDN location). A roundtrip time in a fiber cable takes about 32 ms (assuming 200,000 km/s speed of light in fiber) from the local CDN. Even with this large latency, Facebook discovered that it needed to be aggressive on CDN mapping. In its analysis, Facebook found that only 16% of traffic was coming from local CDNs, and 84% of images/videos were coming from CDNs in South America (literally halfway around the world). For data to travel from South America to Indonesia, it has to travel the length of the Pacific Ocean. If we are generous, and place this CDN in Ecuador (the westernmost tip of South America), your data must still travel ~11,000 miles, giving an RTT of 176 ms (a 5.5x increase!). This undoubtedly shows the value of having a CDN, and additionally the importance of carefully tuning your CDN traffic to minimize the distance/time your data is traveling to your customers!

Testing Your App on Slow Networks

Your first step to testing on slow networks should be a travel request for a world tour of locations where your mobile app is used. (Hey, it doesn't hurt, right?) Facebook, as seen in the previous section, has conducted testing in Africa and Indonesia, and published some interesting results. In Africa, Facebook found that its app burned through a 1-month data plan in 40 minutes. As a result of this trip, Facebook worked aggressively to reduce data usage and network utilization, and with image optimizations and better caching, the Facebook app uses 50% less data (a savings appreciated by all customers, no matter what network speed they are using!).

Few companies have the resources of Facebook, so it is understandable that world travel may be out of the question for app testing. However, with careful analysis of your analytics data, it might be possible to dig through some of these issues by region or country for latency and bandwidth issues that might arise.

Emulating Slow Networks Without Breaking the Bank

In Chapter 2, I suggested using a private Wi-Fi network for your data testing. If your device lab is doing all of its testing on high speed networks, you may be missing important test scenarios on slower networks. By not taking into account the variability of mobile network throughputs, you will potentially alienate customers, and will frustrate existing customers who travel into areas of poor coverage. Carriers use specialized antennas in an isolated environment to test these sorts of situations. But these setups are expensive, so how can you test without breaking the bank? Let's look through these (sorted by cost to implement).

Wi-Fi Throttling

If you are using a Wi-Fi router for your testing, and you can install OpenWRT (an open source router) on it, there is a wshaper plug-in that allows you to throttle the downlink and uplink connections, which at least allows you to emulate slow network speeds (but not the latency).

Emulator

The Android Emulator has the ability to throttle network conditions. When the emulator is open, you can login to the emulator to simulate different throughputs and latencies:

```
telnet localhost 5554
network speed edge  //gprs, umts hsdpa and full are additional options
Network delay edge
```

Homemade Faraday Cage

A Faraday cage is a wire box that isolates the interior space from all external electromagnetic radiation. By building a partial Faraday cage, you can reduce the amount of signal reaching the phone. Some developers have reported success using an old (unplugged!) microwave to partially shield existing radio conditions. The results from these tests might be hard to reproduce due to variability of the experiment, but for qualitative testing this may be sufficient.

Network Attenuator

AT&T has released a tool called the AT&T Network Attenuator, shown in Figure 7-16. The Network Attenuator runs on a Samsung S3 ICS kernel (requires root and flashing of a custom ROM, provided by AT&T). After it's installed, the app works as a dial to slow down the mobile network to lower throughputs (sorry, if you are connected on 3G, it will not speed up your connection to 4G!). When you change the network speed slider from UMTS to EDGE, the uplink, downlink, and RTT timer all adjust, allowing simple testing of your Android app at a slower speed. You can also

adjust the network congestion form left to right, increasing the roundtrip time for each connection.

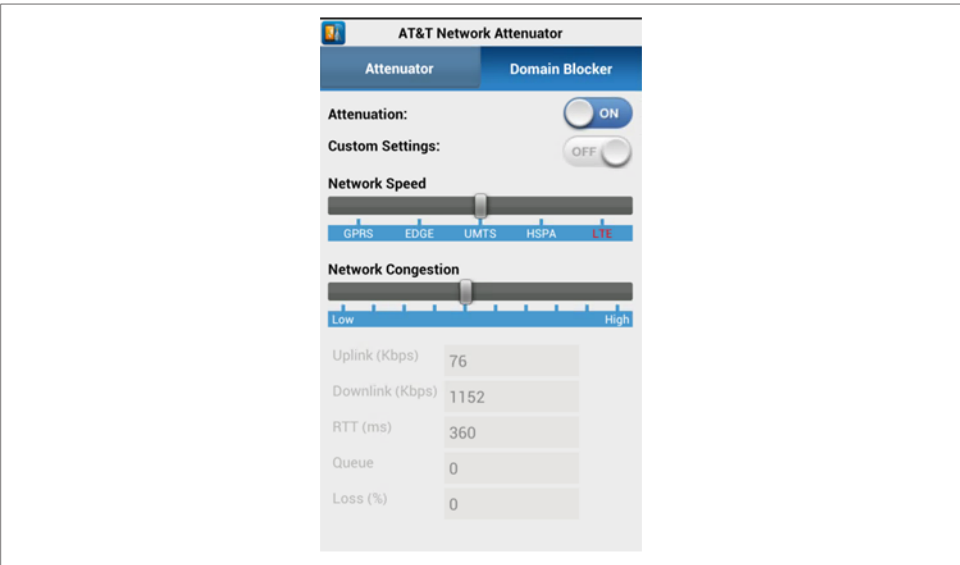


Figure 7-16. Network Attenuator APK

Building Network-Aware Apps

If you know that your customers will be connecting on less than ideal networks (and we know that they will), doesn't it behoove you to ensure the best possible customer experience for them? I am not suggesting that you degrade the quality of your app on 3G or 4G, but there are tricks to enhance the experience on slower networks. I like to call apps that utilize this architecture "flexibly network aware" (FNA), because they are network aware, and flex the user experience based on the measured conditions. Let's walk through the code of my Network Activity Sample app. It is also fun to say the acronym aloud when describing your app.

Imagine that you want to serve a different mobile experience for devices on a fast, medium, and slow network. This could be as simple as removing inline video or reducing the number of images (or at least varying the image size). If the code in Example 7-1 resulted in a cellular connection, you can query the `TelephonyManager`. You can use your app to vary the type of experience to display, as seen in Example 7-4.

Example 7-4. Determining Cellular Network Speed

```
TelephonyManager teleMan =
   (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);
```

```
int networkType = teleMan.getNetworkType();
switch (networkType)
{case 1:    netType = "GPRS";
                        networkSpeed = "slow";
                  break;
case 2:     netType = "EDGE";
                    networkSpeed = "slow";
                        break;
case 3:     netType = "UMTS";
                        networkSpeed = "medium";
                    break;

// we'll leave out a few network types, but you get the idea.
//You can see the full code on GitHub

case 13:    netType = "LTE";
                        networkSpeed = "fast";
                        break;}
```

By querying the network state at regular intervals, your FNA app will gracefully improve or degrade based on the currently available network conditions. This is a pretty basic algorithm, and does not take into account the strength of the network. You could further parse weak signal 3G networks as "slow" or a weak 4G network as "medium."

As an example, you could have a mobile app download a small image on a "slow" network, a medium-sized image on a "medium" network, and a large image on the "fast" network. You can similarly configure Wi-Fi connections as a "fast" network, or perhaps even as a "faster" network, as your customers are not hindered by a cost for each MB of data that you send over Wi-Fi.

Example 7-5. Establishing Network Speed

```
switch(networkSpeed){
case "fast":
        new ImageDownloader().execute(urlbig); //image is 143KB
        break;
case "medium":
        new ImageDownloader().execute(urlmed); //image is 41KB
        break;
case "slow":
        new ImageDownloader().execute(urlsmall); //image is 27KB
        break;
}
```

When it is time to download the images, I calculate the actual time the download is taking, as a quality measurement. I actually record two times: the time until a 200 OK response form the server, and the total time it takes to download the image. The `responsetime` (the time it takes to get a 200 response from the server) is equal to two

RTTs (assuming a DNS lookup has already occurred), and can be used to estimate network latency. The `downloadtime` is the total time it took to receive the object from the server. By additionally querying the content length, my Android app can calculate the actual throughput of the file in KB/s.

Example 7-6. Determining RTT and Throughput in Real Time

```
 private Bitmap downloadBitmap(String url) {
    Long start = System.currentTimeMillis();                //download start time
        final DefaultHttpClient client = new DefaultHttpClient();
  final HttpGet getRequest = new HttpGet(url);
        try {HttpResponse response = client.execute(getRequest);
            //check 200 OK for success
            final int statusCode = response.getStatusLine().getStatusCode();
            //time 200 response received
            Long gotresponse = System.currentTimeMillis();
             }
            final HttpEntity entity = response.getEntity();
            //get ContentLength of file
                        contentlength = entity.getContentLength();
            if (entity != null) {
               InputStream inputStream = null;
               try {
   inputStream = entity.getContent();
   final Bitmap bitmap = BitmapFactory.decodeStream(inputStream);
                 Long gotimage = System.currentTimeMillis();
                                //time image download completed
                 responsetime  = gotresponse - start;
                                //time to the 200 ok response
                 imagetime     = gotimage-start;
                                                        //download time
throughput    = ((double)contentlength/1024)/((double)imagetime/1000);  //KB/s
                 return bitmap;
              }
```

So, what does this data tell us? Using the Network Attenuator app to simulate various network speeds, I was able to measure the download times for the three different files on three different networks, as shown in Table 7-2.

Table 7-2. Download Time(s) of Images

File	LTE	UMTS	EDGE
Big (143 KB)	1.938	5.243	9.405
Med. (41 KB)		2.793	
Small (27 kb)			3.401

If the large file is used for all network conditions (a non-FNA app), it is clear that the user experience on UMTS and EDGE is significantly slower due to the large file size. If we apply a FNA architecture, the UMTS download is nearly 100% faster, and the EDGE download is nearly 300% faster. While using network technology to judge download speeds is (admittedly) a very rough way to estimate the ideal network speed, even this simple model shows the potential for improved customer interactions.

Collecting a database of latency and throughput data on top of network generation and signal strength could allow you to build a better algorithm for allowing your app to flex with the network in near real time, but it appears that keeping it simple still derives benefits to your customers.

Measuring Latency

In my experience, there is a lot of variability in RTT measurements. Distance to the cell tower, congestion, or interference from other radio sources can cause drastic changes in RTT. As such, it is crucial that you not rely on one or two discrete measurements, but instead use a running average to even out any potential outliers. While it is great that you are working with the network conditions at hand, it is important to smooth out this data.

Accounting for Latency

If your FNA mobile app discovers that your customers are in a high-latency environment (due to a calculated high RTT), it can decide to help speed the experience by pre-fetching more aggressively. For example, if you are scrolling through a series of images, you may initiate a download of additional images before the user gets to the end of the list (e.g., if there are only two images below the screen, start getting the next batch of images):

```
if (ImagesBelowtheFold<2){
                <get next batch of images>
                }
}
```

In a high latency environment, your customer might get to the end of the list before the next batch of images are able to load. To account for this, you can begin pre-fetching the images earlier:

```
If (latency = normal){
        if (ImagesBelowtheFold<2){
                <get next batch of images>
         }
}
Else {
         //latency is high
         if (ImagesBelowtheFold<4){
                <get next batch of images>
        //consider getting more images too
        //also, smaller images?
         }
```

By initiating the download twice as early, you are giving the network twice as much time to get the data downloaded before your customer notices a lag. This may use the network slightly more, and potentially download more images (and use more data), so it should be used with caution, but if you can make the user experience seamless, it may be worth it.

Last-Mile Latency

Latency is typically encountered in the *last mile* of transit, and this is especially true in mobile. These tricks can help you *cope* with latency, but they only look to alleviate the problem, not actually solve it. Just as I described in "Testing Your App on Slow Networks" on page 205, Facebook discovered that on slow connections in Indonesia, fully 84% of traffic was being delivered from South America and European CDNs.

"Other" Radios

The cellular and Wi-Fi radios transmitting data are likely the most used, and easiest to optimize. There are additional radios that lead to power drain on mobile devices, and their operation should also be discussed.

GPS

Android offers "Coarse Location" information, that does not require the GPS radio to turn on. By using information about nearby cell towers and Wi-Fi points, a loose location can be generated. However, for many apps, a precise location is needed, and the GPS radio will turn on to receive signals from the GPS satellites. This fix requires a line of sight from your phone to the satellites.

In order to optimize the performance of your location usage, you may have to tweak the window (how long you keep the GPS receiver on), and the frequency. The longer

the window, and the more often the frequency, the quality of your location data will be better.

Bluetooth

Currently, all Android Wear devices must connect to a device via Bluetooth. If you are interested in the traffic sent over Bluetooth, you can collect a logfile that is dissectable in Wireshark. For devices on KitKat and newer, you can enable the "Bluetooth HCI snoop log" under the Developer Options settings menu. When you check this box, your Android device will collect a log of all packets sent along the Bluetooth interface. The data is stored in */sdcard/btsnoop_hci.log*.

Opening this logfile in Wireshark gives you insight into the packets being transferred. Much of data is encrypted, but you can gain insight into the traffic patterns between your two devices (Figure 7-17).

No.	Time	Source	Destination	Protocol	Length	Info
1797	484.650612	controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
1798	484.832305	50:2e:5c:b2:d8:7f (HTC OP6B120)	f8:8f:ca:11:6d:dd (Doug Sillars's Glass)	RFCOMM	1004	Sent UIH Channel=5
1799	484.838119	50:2e:5c:b2:d8:7f (HTC OP6B120)	f8:8f:ca:11:6d:dd (Doug Sillars's Glass)	RFCOMM	861	Sent UIH Channel=5
1800	484.887134	controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
1801	484.890352	50:2e:5c:b2:d8:7f (HTC OP6B120)	f8:8f:ca:11:6d:dd (Doug Sillars's Glass)	RFCOMM	1004	Sent UIH Channel=5
1802	484.895003	50:2e:5c:b2:d8:7f (HTC OP6B120)	f8:8f:ca:11:6d:dd (Doug Sillars's Glass)	RFCOMM	1004	Sent UIH Channel=5
1803	484.896818	50:2e:5c:b2:d8:7f (HTC OP6B120)	f8:8f:ca:11:6d:dd (Doug Sillars's Glass)	RFCOMM	1004	Sent UIH Channel=5
1804	484.898133	controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
1805	484.899369	50:2e:5c:b2:d8:7f (HTC OP6B120)	f8:8f:ca:11:6d:dd (Doug Sillars's Glass)	RFCOMM	1004	Sent UIH Channel=5
1806	484.924935	controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
1807	484.927383	50:2e:5c:b2:d8:7f (HTC OP6B120)	f8:8f:ca:11:6d:dd (Doug Sillars's Glass)	RFCOMM	1004	Sent UIH Channel=5
1808	484.928938	controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
1809	484.930258	50:2e:5c:b2:d8:7f (HTC OP6B120)	f8:8f:ca:11:6d:dd (Doug Sillars's Glass)	RFCOMM	1004	Sent UIH Channel=5

Figure 7-17. Bluetooth traffic in Wireshark

In this case, the response to my query came as a POST, and you can read the response to my Google Query (from Glass) on "Australian Shepherd" (Figure 7-18).

```
[PSM: RFCOMM (0x0003)]
0000  02 05 20 77 02 73 02 4c  00 2d ef dc 04 03 00 00   .. w.s.L .-......
0010  00 14 02 67 50 4f 53 54  20 2f 74 72 61 6e 73 6c   ...gPOST  /transl
0020  61 74 65 5f 74 74 73 3f  69 65 3d 75 74 66 2d 38   ate_tts? ie=utf-8
0030  26 63 6c 69 65 6e 74 3d  67 6c 61 73 73 26 74 65   &client= glass&te
0040  78 74 3d 41 63 63 6f 72  64 69 6e 67 25 32 30 74   xt=Accor ding%20t
0050  6f 25 32 30 57 69 6b 69  70 65 64 69 61 25 33 41   o%20Wiki pedia%3A
0060  25 32 30 54 68 65 25 32  30 41 75 73 74 72 61 6c   %20The%2 0Austral
0070  69 61 6e 25 32 30 53 68  65 70 68 65 72 64 25 32   ian%20Sh epherd%2
0080  43 25 32 30 63 6f 6d 6d  6f 6e 6c 79 25 32 30 6b   C%20comm only%20k
0090  6e 6f 77 6e 25 32 30 61  73 25 32 30 74 68 65 25   nown%20a s%20the%
00a0  32 30 41 75 73 73 69 65  25 32 43 25 32 30 69 73   20Aussie %2C%20is
00b0  25 32 30 61 25 32 30 64  6f 67 25 32 30 64 65 76   %20a%20d og%20dev
00c0  65 6c 6f 70 65 64 25 32  30 69 6e 25 32 30 41 75   eloped%2 0in%20Au
00d0  73 74 72 61 6c 69 61 2e  25 32 30 46 6f 72 25 32   stralia. %20For%2
00e0  30 6d 61 6e 79 25 32 30  79 65 61 72 73 25 32 43   0many%20 years%2C
00f0  25 32 30 41 75 73 73 69  65 73 25 32 30 68 61 76   %20Aussi es%20hav
0100  65 25 32 30 62 65 65 6e  25 32 30 76 61 6c 75 65   e%20been %20value
0110  64 25 32 30 62 79 25 32  30 73 74 6f 63 6b 6d 65   d%20by%2 0stockme
0120  6e 25 32 30 66 6f 72 25  32 30 74 68 65 69 72 25   n%20for% 20their%
0130  32 30 76 65 72 73 61 74  69 6c 69 74 79 25 32 30   20versat ility%20
0140  61 6e 64 25 32 30 74 72  61 69 6e 61 62 69 6c 69   and%20tr ainabili
0150  74 79 2e 26 74 6c 3d 65  6e 20 48 54 54 50 2f 31   ty.&tl=e n HTTP/1
0160  2e 31 0d 0a 55 73 65 72  2d 41 67 65 6e 74 3a 20   .1..User -Agent:
0170  4d 6f 7a 69 6c 6c 61 2f  35 2e 30 20 28 4c 69 6e   Mozilla/ 5.0 (Lin
```

Figure 7-18. Bluetooth POST response

Using Fun Wireshark tricks, you can quantify the packet and data transferred over time over Bluetooth (Figure 7-19).

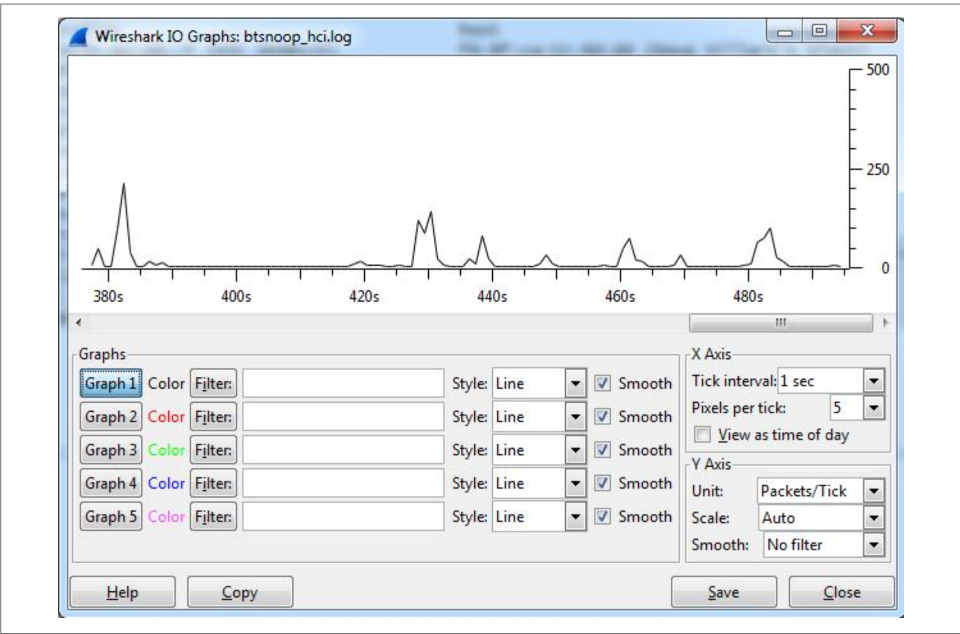


Figure 7-19. Bluetooth POST response - packet counts over time

Conclusion

Most Android apps use the cellular and Wi-Fi radios to communicate to outside servers for information. Because the radios are second to the screen for battery drain, it is important to use them judiciously. Further, as most users have a monthly cellular data allowance, it's doubly important to ensure that the files you are using are optimized for delivery over mobile networks for both speed and the data consumed. In addition to the cost of smartphone data traffic, the higher latencies and slower speeds are essential to account for. Ensuring that your data transfers are optimized for the available network will allow your app to shine, no matter where in the world your customer is, and no matter how good (or bad) the network conditions are. Working to optimize your data traffic to work with the RRC state machine and your customers' location will save customer battery life and enhance the user experience around the world.

CHAPTER 8

Real User Monitoring

In the previous chapters, we have walked through a number of great tools that can be used to diagnose issues in your Android app. We've looked to optimize battery, memory, CPU, and network using free tools that you can use on your Android device. However, as we discussed in Chapter 2 (and you are well aware), these tests require having a physical device in hand, and only permit testing on the Internet connections that are available.

Without a large travel budget and an infinite budget for devices (and unlimited time to focus on performance) how can you ensure that your app is performing optimally for all of your customers, regardless of location, network, or device? The answer is to collect runtime data on your app, aggregate the results, build reports, and look for issues that might arise from the data. These analytics are drawn from the app itself, and is commonly known as real user monitoring (RUM).

While some development teams with deep pockets might build their own RUM engines to gather data, there are a number of tools on the market that you can integrate into your app to collect data from your install base. Many of them are free or have limited free offers, allowing you to begin collecting this information without a huge upfront cost. If your app begins collecting a lot of data or you need detailed reporting, you may have to begin paying for these services, but the value of the data (as you will see) is worth it.

Not Just For Large Teams

Collecting RUM data is not just for the large company with a dedicated performance team. Perhaps you *are* the performance team (along with all the other hats you wear). Collecting data about your users is still very easy to set up in your app, and will provide you with insights to help you improve the usability and performance of your app. I encourage you to give it a shot, and I'm sure you will reap rewards for the small amount of upfront work.

Enabling RUM Tools

There are many RUM tools available in the market. Each will have slightly different reports and data that you might find useful. To gain all of the insights desired, you may find that you need multiple SDKs installed in your app. Each RUM tool provides detailed instructions on how to integrate their code or library into your app. Some have even automated the integration into basically an installer where there is no work involved. More fully featured SDKs allow you to establish your own metrics to monitor and collect data on. In this chapter, I have selected three RUM SDKs to add to a sample app (ImageScroll) to see what data I can obtain.

The first SDK to be added is Crashlytics, which uses a very simple widget (shown in Figure 8-1). All you must do is click the install button, and the code will be automatically added to support these analytics to your app.

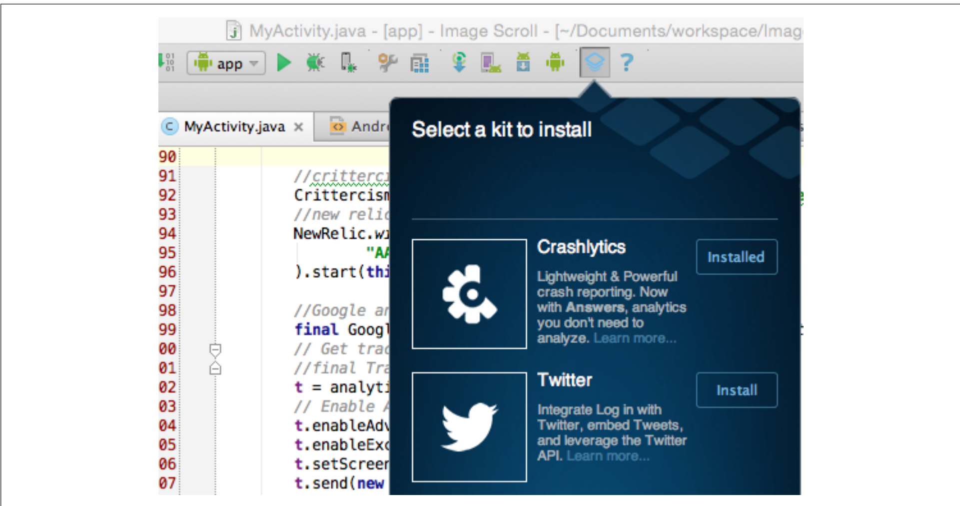


Figure 8-1. Installing Crashlytics

After the SDK is installed, you simply build and distribute your app. As customers begin using your app, the usage statistics are reported back to the RUM provider. Web

dashboards allow you to investigate how customers are using your apps, and where slow downs or crashes might be occurring. By identifying these remotely, you no longer depend on bug reports from your users; you can begin fixing the bugs immediately and release the bug fixes in your next update. This ensures that your app is running optimally on all devices.

RUM Analytics: Sample App

How do these tools collect information on your customers? By inserting either a JAR or library (or sometimes just code), these RUM tools collect information every time the app is run. This data is then transferred to a server that generates dashboards of the data, and can create alerts when things go drastically wrong in your app.

Each tool is different, but many allow you to dissect the data by region, device, OS, app versions, and other criteria. To get sample RUM data, I built an app called "Image Scroll" that simply loads 10 images at a time as you scroll through them (Figure 8-2). The images are hosted on a remote server, and the URLs are stored in an array. When the app reaches the end of the array (there are 92 images), Android throws an out-of-bounds exception, and the app crashes. This is by design, in order to track app crashes across devices.

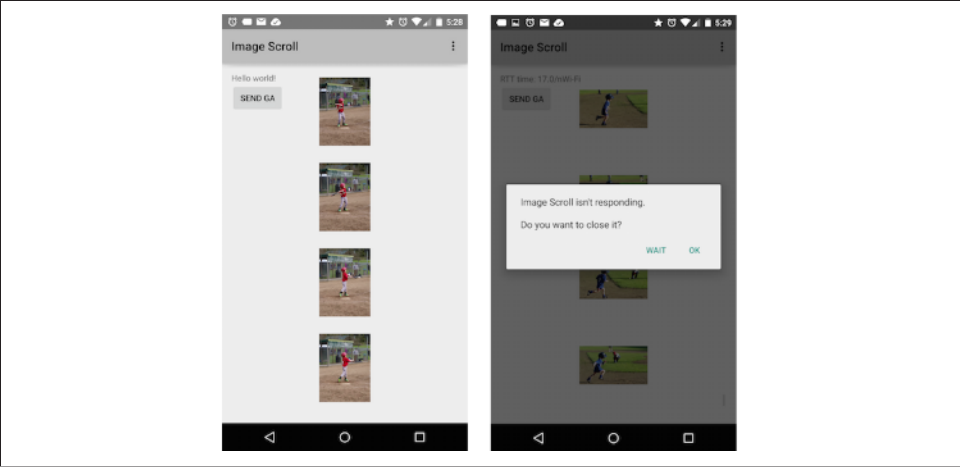


Figure 8-2. Image Scroll app

This app also has several incorrect URLs for the images (to generate 404 errors), and replaces a small image of a baseball player with a large image of goats in the second group (900 KB versus ~50 KB) as test cases for errors that affect performance.

This sample app has (in alphabetical order) Crashlytics, Crittercism, Google Analytics, and New Relic RUM tools installed. I am using free or free trial versions of all of these services. They all report similar information, and each report the data slightly

differently, so one service might work better for your app needs than another. To understand the data being reported, we'll look at screenshots from these tools to better understand the data you can use to optimize your mobile apps.

As we discussed in Chapter 2 when measuring battery drain, there is a bit of a Heisenberg's Uncertainty Principle effect with this RUM data. All of these SDKs will report back to the server with as much information as you want to have reported. This can lead to slightly high data usage and battery drain. These SDKs are fairly well optimized for data usage and battery drain, as we'll see in "RUM SDK Performance" on page 231.

Crashing

As discussed in Chapter 2, performance is crucial to customer satisfaction of your app. Being able to access performance stats on actual customer devices in the field provides you with unprecedented ability to diagnose and resolve issues quickly. Let's start with the most crucial issue when it comes to performance: app stability. Fast notifications of crashes with logfiles can help you quickly diagnose the situation, and fix the issues in your code.

Let's take a look at what happens when we load too many images:

```
        imageViews=new ImageView[100];

    public int Imagelooper
     (int numberofaddedimages, int totalImageCount, RelativeLayout rl){
        for(int i=0;i<numberofaddedimages;i++)
        {

            totalImageCount = totalImageCount++;
            //for analytics I want to track crashes.. so let's force it to crash
           // if(totalImageCount ==100){
           //     totalImageCount=0;
           // }
                   //if totalImageCount reaches 100, the app crashes
                   //because I have exceeded the array size

           imageViews[totalImageCount]=new ImageView(this);
```

When the indexer hits 100, we go out of bounds on the `ImageView` array. Logcat shows:

```
03-13 14:01:32.351  13772-13837/com.sillars.imagescroll I/
  image downloaded: number: 99
03-13 14:01:32.469  13772-13837/com.sillars.imagescroll I/ImageDownloader:
                                 image99responsetime (2RTT): 38
```

We see image 99 downloaded with a roundtrip time of 34 ms. But we are about to increment outside of the array bounds:

```
03-13 14:01:34.637  13772-13772/com.sillars.imagescroll E/AndroidRuntime:
        FATAL EXCEPTION: main
    Process: com.sillars.imagescroll, PID: 13772
    java.lang.ArrayIndexOutOfBoundsException: length=100; index=100
            at com.sillars.imagescroll.MyActivity.Imagelooper
              (MyActivity.java:327)
            at com.sillars.imagescroll.MyActivity$3.onScrollStopped
              (MyActivity.java:178)
            at com.sillars.imagescroll.MyScrollView$1.run(MyScrollView.java:37)
            at android.os.Handler.handleCallback(Handler.java:739)
            at android.os.Handler.dispatchMessage(Handler.java:95)
            at android.os.Looper.loop(Looper.java:135)
            at android.app.ActivityThread.main(ActivityThread.java:5221)
            at java.lang.reflect.Method.invoke(Native Method)
            at java.lang.reflect.Method.invoke(Method.java:372)
            at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run
                        (ZygoteInit.java:899)
            at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:694)
```

And indeed, we get an out-of-bounds exception for the `ArrayIndex` (length is 100, and index set to 100):

```
03-13 14:01:35.329  13772-13797/com.sillars.imagescroll I/Fabric:
                Crashlytics report upload complete: 35CC-27DD9B9DA026.cls
03-13 14:01:54.291  13772-15861/com.sillars.imagescroll I/
   com.newrelic.agent.android: Harvester: connected
03-13 14:01:54.291  13772-15861/com.sillars.imagescroll I/
   com.newrelic.agent.android: Harvester: Sending 102 HTTP transactions.
03-13 14:01:54.291  13772-15861/com.sillars.imagescroll I/
   com.newrelic.agent.android: Harvester: Sending 1 HTTP errors.
03-13 14:01:54.292  13772-15861/com.sillars.imagescroll I/
   com.newrelic.agent.android: Harvester: Sending 0 activity traces.
03-13 14:02:05.070  13772-13772/com.sillars.imagescroll I/Process:
                        Sending signal. PID: 13772 SIG: 9
```

After the app has the exception, we note that there are a few more log entries after the crash—these are the crash reports being pushed up to Crashlytics and New Relic. I have noticed (using network monitoring) that Crittercism reports generally occur a short time after the app was exited.

It is great to be able to reproduce an error in a controlled test environment where we have the phone and analysis equipment. Because that will not always be the case, let's see what these tools report to us. All of the apps report crashing in a similar way, so let's looks at some sample reports from Crashlytics.

Examining a Crashlytics Crash Report

When Crashlytics finds a new crash, you immediately get an email reporting that a new issue was found (hint: build a special *crash report* email address or filter). Clicking the link in the email takes you to the crash details web page. Figure 8-3 shows a screenshot of the Imagelooper crash.

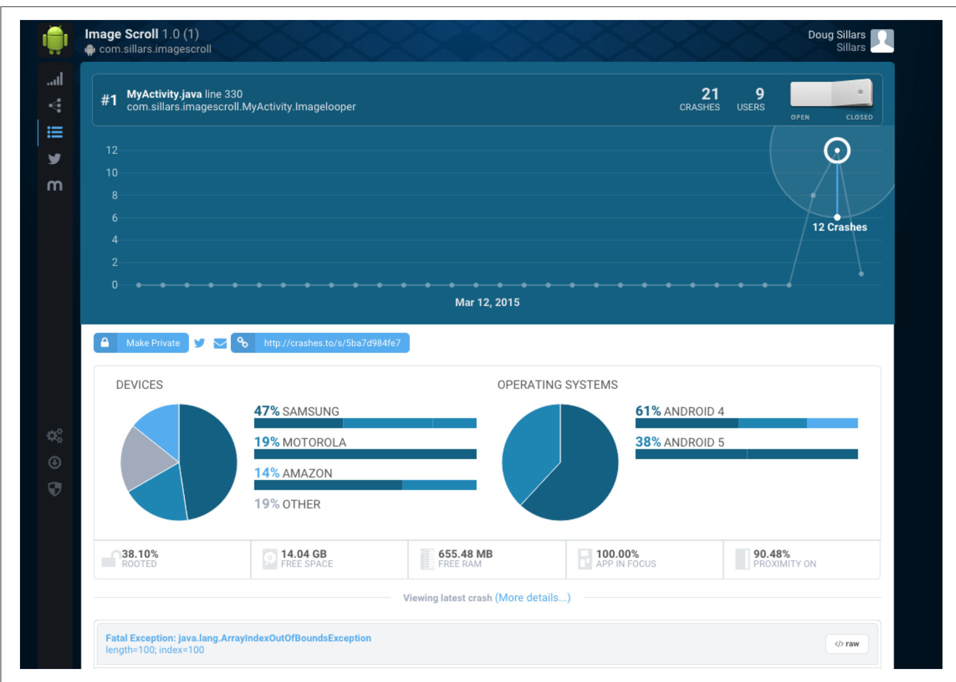


Figure 8-3. Crash details

The top of this dashboard shows the number of crashes and the number of users (in this case, 21 crashes across nine users), with a graph showing the count of crashes over the last three days (the 12 crashes from March 12, 2015, are highlighted). The pie and bar charts in the middle of the page breaks down the devices where the crash has occurred by OEM (left) and Android version (right). Clicking on either chart gives a further breakdown (in this case, we are looking at the count of Samsung devices and devices running Android 4 variants), you can break down devices further (and the same with the OS versions); see Figure 8-4.

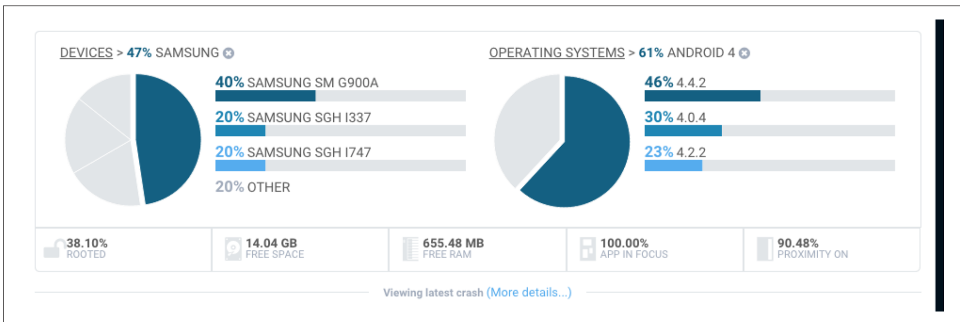


Figure 8-4. Device breakdown of crashes: focusing on Samsung devices and Android 4 OS versions

The section underneath the pie charts gives average device details during crash: were the devices rooted? Was there free disk space/RAM? Was the app in the foreground? Was location on? Below this is the exception trace matching what we saw in the logcat. You can interact with the latest crashes, and get exact details about the exact device (including all active threads) through the interface. I have shared this crash publicly, so you can examine these details further if you are interested: *http://crashes.to/s/5ba7d984fe7*.

The availability of the remote trace of the crash makes debugging a bit easier. All of the tools offer either bug tracking in their interface, or a way to export the defects from their system into a bug tracking repository. While being reactive to crashes is not ideal, these tools allow you to see which crashes are affecting the most customers and prioritize their resolution.

Assuming you've tackled all of the pressing crashes reported in the previous section, now you can investigate how your app is performing around the world. Tools that report how your app performs on different devices and networks around the world can help you isolate issues, and find bottlenecks in your app that traditional testing might not find. Perhaps your app crashes on a popular device in the Middle East, or you suddenly have a lot of usage on a slower network in Africa. Or perhaps these tools will pick up unexpected usage patterns in your app that you can capitalize on for future releases. There are a number of SDKs and tools tha report this information. Figure 8-5 a report of app usage in the last 24 hours from Crittercism.

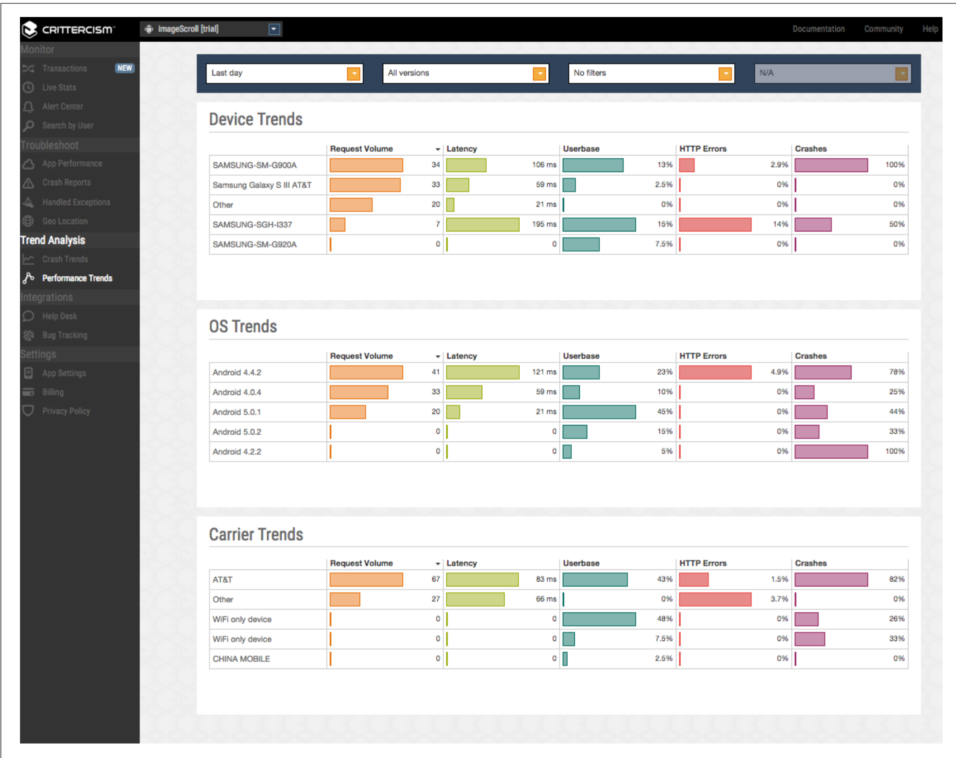


Figure 8-5. Crittercism report: 24-hour usage trend

This dashboard breaks down the number of requests (orange), latency in ms (green), userbase % (blue), HTTPS errors (red) and the crash % (purple) for the top devices (top), OS versions (middle) and carriers (bottom). From this chart, it appears that most of the usage has been from Samsung devices with varying latencies, errors, and crashes per device. This allows you to quickly see if your app is having trouble with a specific device or OS version. If a group of devices, (or OS versions) are exhibiting excess latencies, or crashes, you can investigate and push out fixes for those users.

The wireless carrier charts work as a rough proxy for the location of your users. There is also a global map showing users by country in all of these tools, but because this data is all generated by me—and one helpful person on Twitter from Shanghai—the maps are pretty boring.

That one connection from Shanghai is interesting though. It appears that the connection was very slow, and there were a number of errors that occurred during the transaction. Here is the dashboard from New Relic (and the connection occurred on March 12 just after midnight Pacific Time). In the main chart on the page, we see that this one connection takes nearly 20 s (and all the other connections nearly do not render on the page). The color under the graph indicates that a network issue caused

the problem. By observing how your app is behaving over time, you can discover network or server delays during busy traffic times, indicating that the server is overloaded, or that the files might be overtaxing the network.

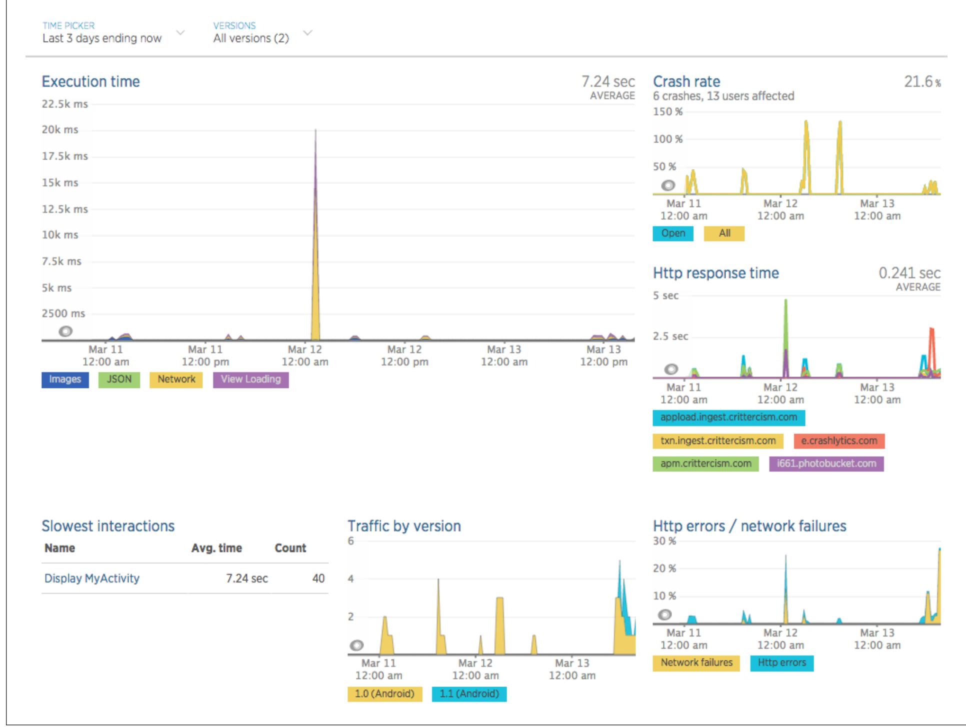


Figure 8-6. Dashboard with one slow connection

The execution time of this one connection from China is well above the rest. It could just be a random outlier. But if I continued to see slow connections to one area, it would justify further investigation. The middle graph on the right of this dashboard is the HTTP response time. For the connection to China, the Crittercism response time is nearly five seconds (again, not customer facing, so that's OK), but the Photobucket response time is 1,720 ms. This dashboard also includes time graphs for the crash rate (where the color matches the bug status), traffic by app version and HTTP errors over time.

Some tools also allow you to see latency by domain, as shown in Figure 8-7 (are your servers doing OK?). Filters for carriers allow you to see if certain regions might not be getting the data fast enough (recall from "Testing Your App on Slow Networks" on page 205 that Facebook found issues specific to certain countries). If you see these sorts of trends, you can instrument your app further in an attempt to identify the slow points). In the chart below, the latency peaks with an average of 200 ms, but

deeper investigation shows that the slower connections are analytics, while my image provider (Photobucket) has a response time of 23 ms. The error rate is due to the fact that 2 of the 100 images loaded by my app have incorrect URLs, throwing a 404 error. Seeing any error here should prompt you to dig deeper.

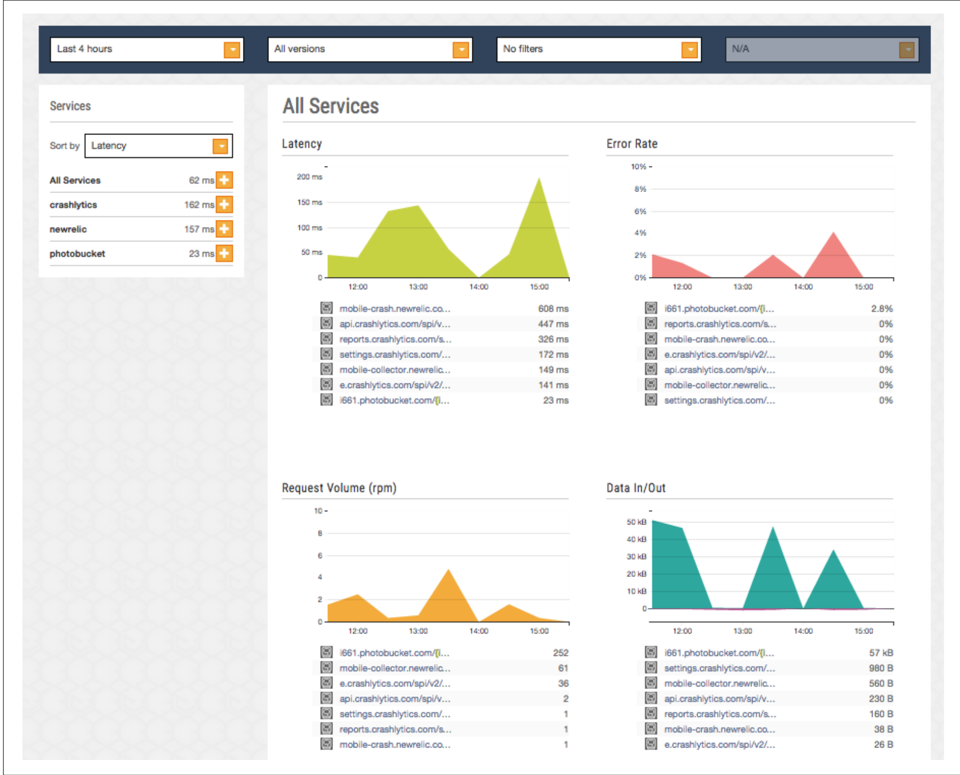


Figure 8-7. Latency, errors, volume and data graphs

The New Relic tools have a list of all network errors. In Figure 8-8, we can see all of the issues from Photobucket (the host of the images) are 404 errors. Clicking into this section provides a list of all 404 errors from the Photobucket domain—again giving you the opportunity to resolve these issues on the backend.

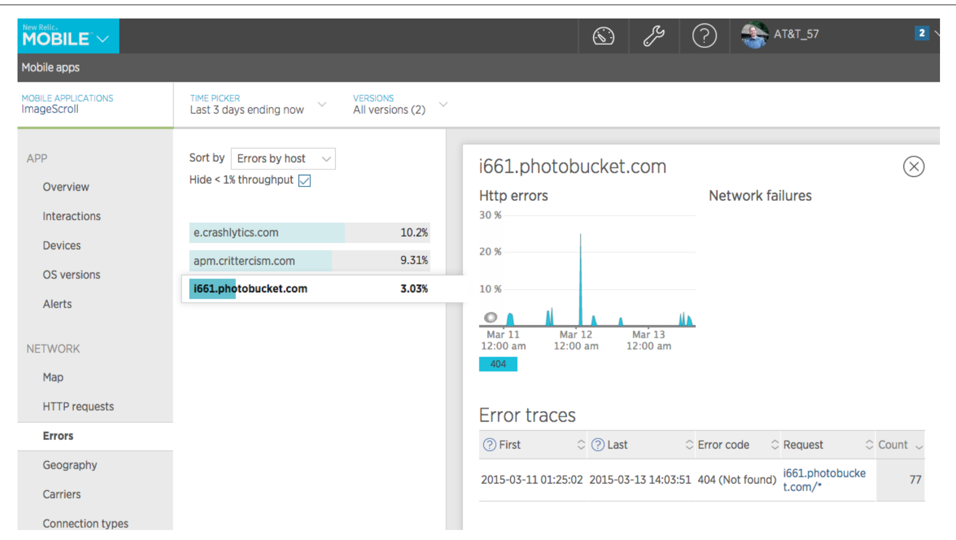


Figure 8-8. Network errors

Being able to correlate HTTP errors from your app with your server logs allows for very powerful troubleshooting. You can narrow down the issue to a platform, or version of your app that might be causing troubles.

Usage

Beyond crashes and performance, your RUM data can also tell you a great deal about your customers: how they are using the app (and how long), how often they use it, and more (Figure 8-9). By better understanding who your users are, and how engaged they are, you have more opportunities to improve your app.

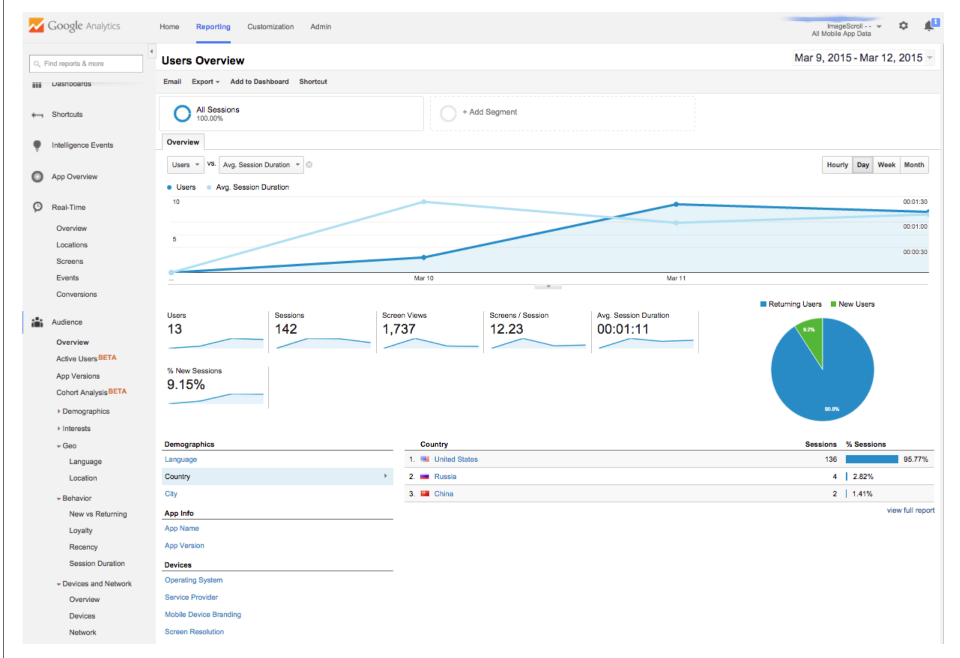


Figure 8-9. Usage information

The Google Analytics data provides us with a lot of information about our users. The top graph is showing the daily user count versus the average session length (on March 11, there were nine users with an average session time of 59 seconds). There are 13 total users with 142 total sessions (91% of sessions are from returning users according to the pie chart). Google Analytics allows you to specify specific screens that appear to the users. From this data you can gather information about what views in your app lead to customers exiting, or new user flows that you can improve. For an app with just one screen, I label every 10th unique image as a screen:

```
//initialize tracker at top of the screen
t = analytics.newTracker(R.xml.app_tracker);
        // Enable Advertising Features.
        t.enableAdvertisingIdCollection(true);
        t.enableExceptionReporting(true);
        t.setScreenName("top of scroll");
        t.send(new HitBuilders.ScreenViewBuilder().build());
<snip>
//10 more images were just requested, so update the screen name in Google Analytics
                    t.setScreenName(totalImageCount + " images");
                    t.send(new HitBuilders.ScreenViewBuilder()
                            .build());
//and add a crittercism Breadcrumb
                 Crittercism.leaveBreadcrumb(totalImageCount + " images");
```

After adding these sections of code to my app, whenever a customer scrolls through 10 images in it, Google adds a screen view and Crittercism adds a breadcrumb for issue tracking. Let's look at the dashboard report from Google Analytics (Figure 8-10). This table shows what you might expect, that page views at the start of the app are most common ("Doug Scroll App" is the initialized name of the screen), and then users exit at various times as they scroll through the app. A large percentage of the exits occur without any scrolling, and also there are a lot that exit at 90 images (due to the bug that causes the app crash after image 99). Another interesting feature is that the average time in a view is highest for 90 images. I have seen the app go to an ANR rather than crash, freezing the app into a holding pattern, and inflating the usage time for this screen.

Primary Dimension: Screen Name

Plot Rows | Secondary dimension | Sort Type: Default | advanced

	Screen Name	Screen Views	Unique Screen Views	Avg. Time on Screen	% Exit
		1,737 % of Total: 100.00% (1,737)	702 % of Total: 100.00% (702)	00:00:06 Avg for View: 00:00:06 (0.00%)	8.12% Avg for View: 8.12% (0.00%)
1.	10 images	283 (16.29%)	15 (2.14%)	00:00:01	0.00%
2.	20 images	283 (16.29%)	77 (10.97%)	00:00:03	2.12%
3.	Doug Scroll App	223 (12.84%)	137 (19.52%)	00:00:13	17.04%
4.	30 images	180 (10.36%)	68 (9.69%)	00:00:07	0.00%
5.	40 images	142 (8.18%)	60 (8.55%)	00:00:08	1.41%
6.	top of scroll	95 (5.47%)	79 (11.25%)	00:00:00	65.26%
7.	50 images	85 (4.89%)	56 (7.98%)	00:00:07	4.71%
8.	60 images	79 (4.55%)	52 (7.41%)	00:00:04	3.80%
9.	70 images	64 (3.68%)	49 (6.98%)	00:00:11	6.25%
10.	80 images	63 (3.63%)	42 (5.98%)	00:00:03	1.59%
11.	90 images	53 (3.05%)	40 (5.70%)	00:00:34	24.53%

Figure 8-10. Screen views in app

For a real app, the time spent in a unique view can tell you a lot about how your customers are interacting with your app. In addition to graphing the time in each screen, Google Analytics can break down the flow from one screen to the next. In this simple app, it makes sense that most users will scroll from 10 images to 20 images, etc. For a more complex app, this can help you find issues with your flow, or views that your customers are not finding. If you observe devices or screen sizes that are missing screens, perhaps there is an issue with the way the clinks are rendering—inhibiting the customers from browsing your app as expected. The flow data can be broken down in many ways: from all users to smaller subsets. In Figure 8-11, all of the data traffic is in gray, while the darker lines indicate the data traffic just for users from California.

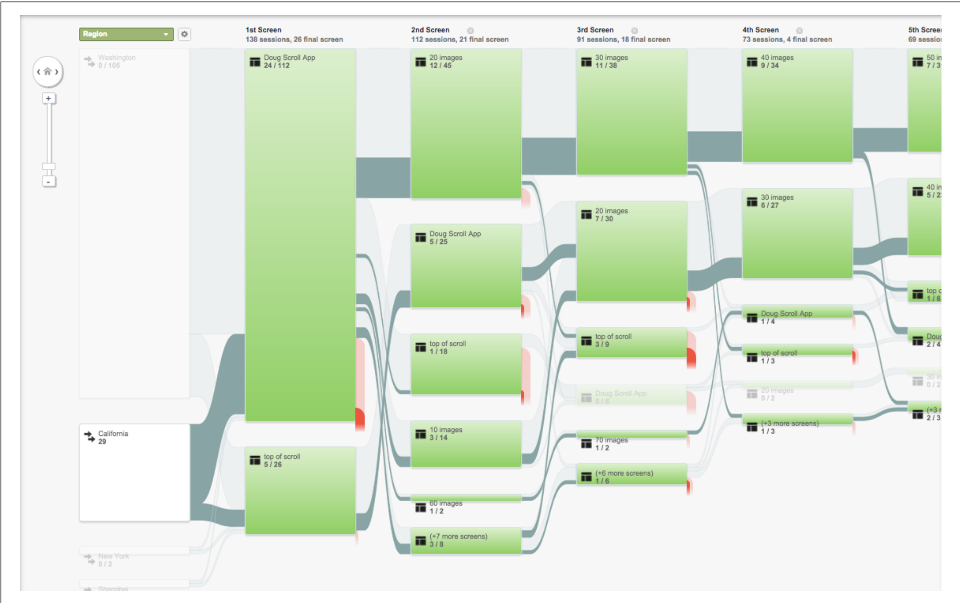
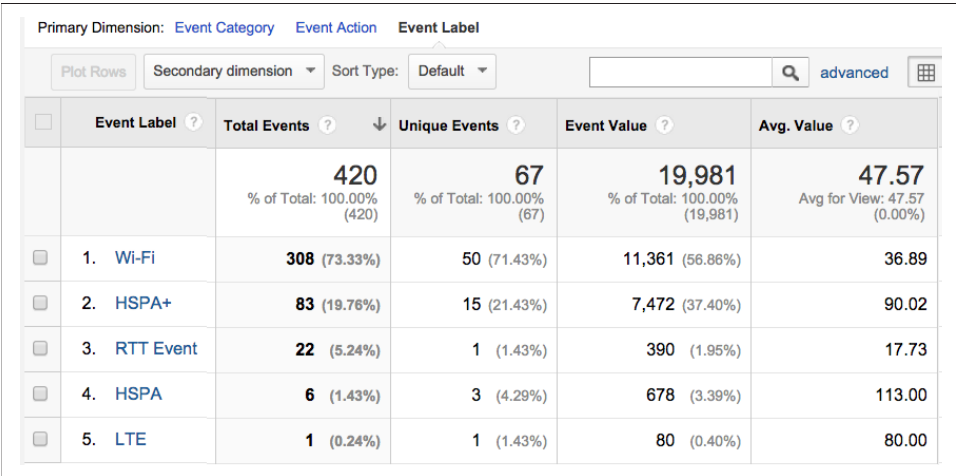


Figure 8-11. Behavior flows (California users are highlighted)

You can find other pain points by setting unique events that are reported back to the analytics server. In Chapter 7, we used Example 7-4 to identify the Network type used by the customer and Example 7-5 to modify the content delivered based on the available network bandwidth. I have applied this logic to the Image Scroll app, and additionally report the network type and the RTT times I am finding to the analytics engines:

```
t.send(new HitBuilders.EventBuilder()
              .setCategory("RTT Event")
              .setValue(AvgRTT.longValue())
              .setAction("ImageRTT").setLabel(networkConnection).build());
Crittercism.beginTransaction(networkConnection);
Crittercism.setTransactionValue(networkConnection, AvgRTT.intValue());
Crittercism.endTransaction(networkConnection);
```

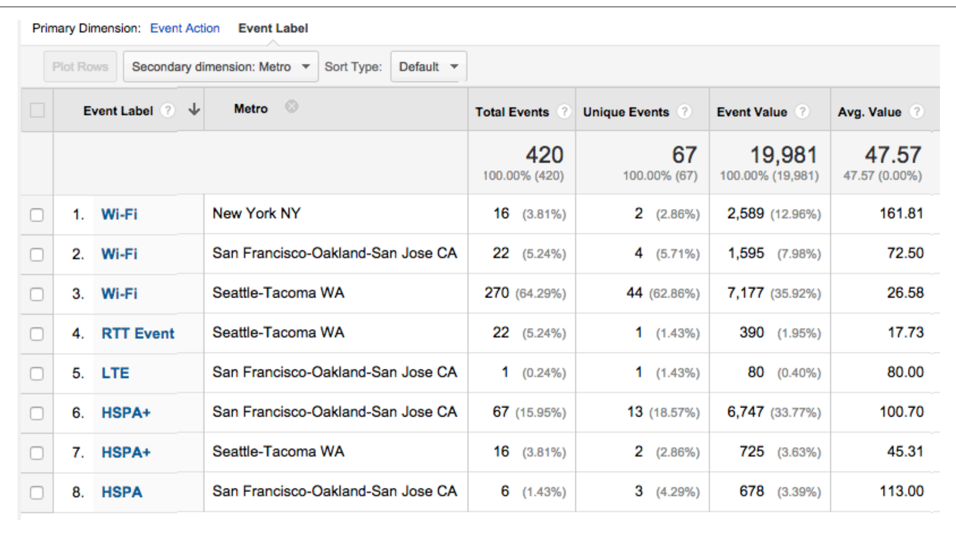
This data is reported as shown in Figure 8-12.



	Event Label	Total Events	Unique Events	Event Value	Avg. Value
		420 % of Total: 100.00% (420)	67 % of Total: 100.00% (67)	19,981 % of Total: 100.00% (19,981)	47.57 Avg for View: 47.57 (0.00%)
1.	Wi-Fi	308 (73.33%)	50 (71.43%)	11,361 (56.86%)	36.89
2.	HSPA+	83 (19.76%)	15 (21.43%)	7,472 (37.40%)	90.02
3.	RTT Event	22 (5.24%)	1 (1.43%)	390 (1.95%)	17.73
4.	HSPA	6 (1.43%)	3 (4.29%)	678 (3.39%)	113.00
5.	LTE	1 (0.24%)	1 (1.43%)	80 (0.40%)	80.00

Figure 8-12. Network type seen by app

The network type was checked on every screen update, and thousands of RTT times were collected (column 3 shows that nearly 20,000 values were obtained). The average roundtrip time is reported in the last column of Figure 8-12. The average roundtrip time might not be extremely useful, as it varies by signal strength, location, and the type of network. However, using a secondary dimension to the data, we can get RTT by network type by device, metro area, continent, allowing slicing and dicing of the information in many different ways. Figure 8-13 sorts the data by U.S. metro region, showing that the Wi-Fi in Seattle has a faster RTT than that in San Francisco and New York.


Primary Dimension: Event Action Event Label

Plot Rows | Secondary dimension: Metro | Sort Type: Default

	Event Label	Metro	Total Events	Unique Events	Event Value	Avg. Value
			420 100.00% (420)	67 100.00% (67)	19,981 100.00% (19,981)	47.57 47.57 (0.00%)
1.	Wi-Fi	New York NY	16 (3.81%)	2 (2.86%)	2,589 (12.96%)	161.81
2.	Wi-Fi	San Francisco-Oakland-San Jose CA	22 (5.24%)	4 (5.71%)	1,595 (7.98%)	72.50
3.	Wi-Fi	Seattle-Tacoma WA	270 (64.29%)	44 (62.86%)	7,177 (35.92%)	26.58
4.	RTT Event	Seattle-Tacoma WA	22 (5.24%)	1 (1.43%)	390 (1.95%)	17.73
5.	LTE	San Francisco-Oakland-San Jose CA	1 (0.24%)	1 (1.43%)	80 (0.40%)	80.00
6.	HSPA+	San Francisco-Oakland-San Jose CA	67 (15.95%)	13 (18.57%)	6,747 (33.77%)	100.70
7.	HSPA+	Seattle-Tacoma WA	16 (3.81%)	2 (2.86%)	725 (3.63%)	45.31
8.	HSPA	San Francisco-Oakland-San Jose CA	6 (1.43%)	3 (4.29%)	678 (3.39%)	113.00

Figure 8-13. Network type by city

By naming your screens, adding custom events and timers, you can build a very detailed picture of how your customers are using your app, find pages that are loading slowly, navigation points that are being missed, and other user flow issues. You can also discover if certain regions of the world are facing higher latency, errors, or other slowdowns. You can discover if there are performance issues on specific days, or times during the day due to congestion on the network (or even on your server!).

Real-Time Information

As the analytics data being collected by these reporting SDKs is being sent regularly, you can track users in near real time. All of the providers just discussed show the performance of apps in near real time. In Figure 8-14, we see four app loads (blue line), and one crash (red line) in the last 30 minutes.

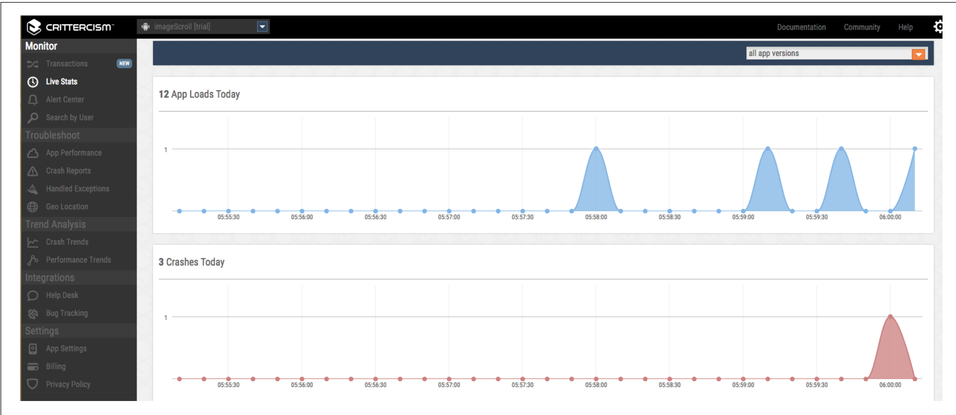


Figure 8-14. Real-time analytics

Big Data to the Rescue?

As your app grows and gains a larger user base, the ability to quickly ascertain and resolve issues becomes even more important, but also more difficult. Figuring out which issues are affecting the most customers and the severity of the problem helps you to prioritize bug fixes. The tools described in this chapter can help you crunch the numbers and find the usage patterns and issues that require optimizing—streamlining your app and making your customers happier.

Here is where good RUM will help you ensure that updated releases are performing better than previous versions, reducing crashes and improving speed and load times. While using the big data collected by your RUM tools is still a reactive way to resolve issues, careful planning will provide you with the insights you need to continuously improve your app, and report how performance improvements actually improve the retention and time spent in the app.

RUM SDK Performance

Despite the fact that these SDKs are built to measure performance, it is a good idea to measure the performance of these tools. If you notice in the previous screenshots, each SDK reports the latencies of the *other* SDK connections (but not their own). Short roundtrip times are important for user critical data, but longer roundtrips for files to be accessed later are OK. Should you begin to see a lot of HTTP connection errors form the SDK, then you might begin to worry.

To test the network performance of your monitoring SDKs, you can use a tool like the Application Resource Optimizer "AT&T Application Resource Optimizer" on page 183. ARO has tools that allow you to filter packet data based on endpoint.

Figure 8-15 shows us the full app data on top, and just the analytics data view on the bottom.

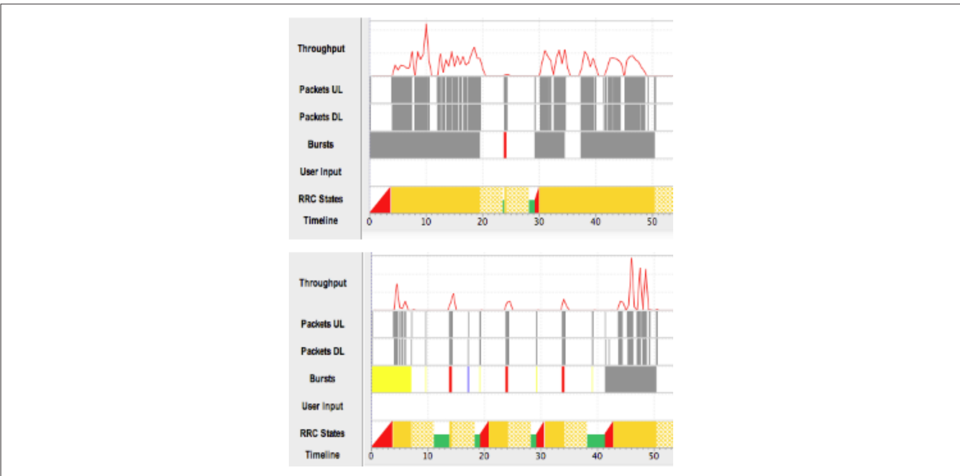


Figure 8-15. Network trace of image Scroll: full app (top) and just RUM connections (bottom)

The RUM files from these providers are all encrypted, so to view them, I ran the same test through a "MITMProxy" on page 183. The RUM data being sent by these analytics providers does not endanger any customer data, and none of the data being sent is unexpected. Here is a crash report from Crittercism:

```
2015-03-16 13:38:41 POST https://api.crittercism.com/android_v2/handle_crashes
                         ← 200 application/json 14B 46.33kB/s
Request
Response
x-newrelic-id:     <removed>
Accept:            text/plain
Accept:            application/json
Content-Type:      application/json
User-Agent:        5.0.6
Host:              api.crittercism.com
Connection:        Keep-Alive
Accept-Encoding:   gzip
Content-Length:    28095
JSON
{
    "app_id": "<removed>",
    "crashes": [
        {
            "app_state": {
                "activity": "com.sillars.imagescroll.MyActivity",
                "app_version": "1.1",
                "app_version_code": 2,
```

```
"arch": "armv7l",
"battery_level": 0.16,
"carrier": "",
"disk_space_free": "11018395648",
"disk_space_total": "24723058688",
"dpi": 3.5,
"locale": "en",
"memory_total": 268435456,
"memory_usage": 90950656,
"mobile_country_code": 0,
"mobile_network": {
    "available": true,
    "connected": false,
    "connecting": false,
    "failover": false,
    "roaming": false
},
"mobile_network_code": 0,
"model": "Nexus 6",
"name": "",
"orientation": 1,
"sd_space_free": "11018395648",
"sd_space_total": "24723058688",
"system": "android",
"system_version": "5.0.1",
"wifi": {
    "available": true,
    "connected": true,
    "failover": false
```

We can see the app, version, that my battery was pretty low (16%), that there is a lot of free disk space and memory, that I was on Wi-Fi (but cellular was available), and a lot more. All of this data is used on the dashboard to help diagnose the crash, and in all of the files collected, there is no unexpected data being transmitted.

Conclusion

In this chapter, we looked at how collecting RUM analytics data from your customers can help you ascertain issues on devices you are unable to test on. By getting log traces of crashes on these devices, it is possible to resolve the issue without every handling the device in question.

The data you collect can also help you uncover regional pain points by noting slow network connections in certain areas of the world. By carefully tracking user behavior, you might find that your customers are using your app in unexpected ways, and in order to accommodate these new uses, streamline the user flow in order to make the experience better.

By carefully instrumenting your app with analytics, you can obtain very powerful data on how your app is failing, where it needs improvement, and where things are running well with real user monitoring. Getting real data form the field on real devices with real customers is invaluable, and when analyzed carefully, can be used to great advantage—fixing all of the problem points that you did not catch with your tests run in the lab.

APPENDIX A

Organizational Performance

In order to be successful in instituting performance in all aspects of your Android app, it helps to have your entire organization "buy in" to the importance of performance. Developers, testers, and your management all need to agree that ensuring fast performance is crucial for the app (and maybe your company's success). Once the team is on board, you'll need to develop processes to ensure that performance remains a metric that your development and apps are held to. Finally, we'll review a number of the tools outlined in this book as a part of your performance process implementation.

Getting Buy-In (Management Focus on Performance)

When it comes to making app performance a part of your company's culture, it is crucial that you gain the buy-in of your management. There is a lot of data out there on slow website performance, and the little data I have seen on mobile app performance follows the same trend. So, if you are having trouble convincing your management that app performance is essential to the lifeblood of your company, refer back to "Performance Matters to Your Users" on page 2 and "Performance Infrastructure Savings" on page 4 for the data points that might sway your leadership—and what company is not looking to lower costs or increase revenue? Perhaps a case study from the *New York Times* on how slow performance was a factor in sinking Friendster (a social media pioneer) might help.

Tying this information up with potential issues and proposed app optimizations is often a great way to initiate a conversation in performance. As fixes are made, and gains are seen in usage, user engagement, and sales, the case to expand ongoing performance is an easier task. If you are the first person in your organization, you will probably start off as the person testing and discovering issues.

Steve Souders's post on "Creating a Performance Culture" has an important point on speaking the vocabulary of your audience. If you are trying to win over marketing, talk about increasing users, engagement, and sales. Operations wants to hear about changes in capacity or outage reduction, while finance would love to hear about increases in sales while reducing costs. By slightly changing your pitch to the keywords of the listener, we have found that buy-in comes more easily.

At AT&T, we have been extremely lucky when discussing performance with our leadership. They realize that having high-performing mobile apps leads to less data usage, longer battery life, and ultimately, to happier customers. As a result, AT&T has instituted performance testing requirements for all internally developed apps, all apps that are preloaded on our devices, and we continue with outreach with developers both inside and outside the company.

Talking About Performance

In early 2011 (think mid-Gingerbread era), we began working on AT&T's Application Resource Optimizer. We were beginning to look at how Android apps used data, and were surprised at how inefficient they were. As we began speaking with developers, we realized that no one was really thinking about mobile data performance. We found that the 80/20 rule really held in this case. 80% of the time, if developers had awareness into app behavior, they would work to optimize it. The other 20% might be stuck with obstacles; perhaps organizational, or requiring more help with testing.

When presenting performance issues to other teams, always use the carrot over the stick. No one appreciates being called out, especially on something that was not on their radar as a concern. If you can point to potential speedups or improvements of the app, stick to the positive. You may gain additional followers on the path of performance.

Lara Hogan writes about becoming your team's performance cop/janitor, and how that can lead to burn out. She argues that a performance lead is essential, but they should work to institute processes around the company that make performance part of the daily routine. As a part of an outreach team that talks about performance, we do sometimes act as the watchdog for the companies we work with. The developers and managers in these companies know about performance, and they test for it, but sometimes, with all of the many burdens placed on the developers, the performance testing falls by the wayside. Simply sending them a friendly reminder every few months about performance keeps them on track.

At Oredev in 2013, Scott Barber conveyed a story about a project that had no budget for optimization or performance testing. He asked the front desk admin to ask the developers once a week about the performance of the app, and he found that a simple reminder about performance kept it *on their mind* during development, and helped

to reduce load speeds. Read blogs about performance, and share tidbits with your colleagues. When you discover a new performance technique, share it, both inside your organization and outside. By helping others learn how to make mobile faster, you excite and energize your team and those you work with.

By making performance a part of the regular conversation in your organization, you'll begin to regularly find performance gains and wins. Speak often about performance. Share the successes other teams (outside your company) have shared, and also share big wins inside your organization. There will be setbacks. Apps will launch with issues. The trick is to quickly identify the problem and resolve it as quickly as possible. The following sections cover a few strategies that we have found to be successful.

Development

We all know the maxim/joke: the best code is no code at all. As soon as the first code hits the screen, our app is slower than it was a moment before. As code is changed, improved, or added to, the change in customer performance should be considered, developed to be minimized, and finally tested.

If your developers are thinking about performance, they will work to ensure that each new feature is built in a way to seamlessly add the new features. This doesn't always happen. Even working on a product to test app performance, the AT&T Application Resource Optimizer team has found stories built without taking heed to performance.

We were adding a new best practice to the ARO tool, and the developer did not work to integrate the story into existing code, but just tacked on addition code. The result was that the app scanned the multimegabyte network trace multiple times rather than once. The performance time for analysis increased dramatically.

As a result of this development snafu (on a performance tool no less!), we began a path of looking at the performance of every change made to the app. I wish I could say this is all automated, or that we use a stopwatch to time the differences in performance when changes are implemented, but that's not the case. We do compare the code whenever additions are made to ensure that the performance costs are acceptable. In our team, the story owners work very closely with developers, and often get to see rough versions of tools and features, allowing us to comment on UI, layout, syntax, and (of course) performance.

Inside AT&T, the ARO outreach team acts as a support team for performance issues. We have gotten requests for help from many different organizations inside AT&T for both internal and external apps. By helping these developers and showing the common pitfalls, they are often able to quickly resolve the issues that are slowing down their Android app. Having the development team understand and be empowered to

research and fix performance issues can be a challenge (with all the new features, bugs, and technical backlogs, it is tough), but correcting the crucial performance issues can really help the bottom line.

Testing

Yes, testing. Always one of the first things to be cut when a deadline looms and the schedule starts getting compressed. Without performance testing, you may only discover that a new feature slows your app through your analytics. But now all foxes are reactive to customers, as you have exposed a slowdown in production. As I described earlier "Performance as a Rolling Outage" on page 6.

When a new feature is added, there are undoubtedly tests that are run to make sure that it works as expected, and does not break other parts of your app. If you only test for crashes, you are handling the *outage* performance, but make sure that new code is not adding latency or slowdowns as well. They can cost your app just as much as the crashes. If the new feature causes a slowdown beyond an expected amount, a process should be undertaken to determine if the change in speed is acceptable, or if the feature should be sent back for further optimization.

Testing Tips

AT&T, in partnership with the Application Quality Alliance (AQuA), has come up with a number of best practices for testing network performance. The test cases are a good starting point to beginning a performance test suite. If you have great performance test cases, share them with me, and we will work to share them with other developers.

Performance Metrics

When it comes to performance, it is up to your team to determine the correct metrics for speed. There are many studies on what customers expect from the Web (and studies focusing on apps are increasing). Do your own testing, and see what your users expect, and if they find your app slow. If you find that your app is slow, determine what reference devices are slower than others in your device lab (Chapter 2), and test with these during development.

Mobile apps are so varied and unique that building a "go to" test case that holds for all apps is next to impossible. Cases that are essential for streaming apps will not hold for social apps, games, or news apps. The test cases I shared here are extremely generic, and could easily be tightened for a specific app. For your apps, work with your team to come up with common sense requirements and then codify them so that they stick.

When a metric is surpassed, make sure that a bug is created, and that the issue is resolved as quickly as possible (ideally before it is released).

Testing Your Performance Metrics

The most frequent complaint customers voice about their Android phones is battery life. In the past, the blame was focused entirely on the manufacturer of the device, or a faulty battery, but customers are becoming savvy to apps causing issues too. In Chapter 3, we looked at ways to measure the battery drain your app causes, from wakelocks to overuse of the device's radios. We examined the Lollipop JobScheduler API as a possible resolution for newer devices, and how using the Battery Historian can pinpoint issues in your app that are causing battery drain.

Your customers interact with your app on the screen, and slow or janky scrolling is a prime factor in app abandonment. Often, it is simply the perception of speed in your app that affects your app usage, and in Chapter 4 we looked at how to simplify UI hierarchies and test your UI for jank and speed issues with Systrace and other tools. If your app is suffering from crashes due to memory leaks or "not responding" issues, tools such as MAT or Traceview (discussed in Chapter 7) will help you figure out what is causing the issue, so that you can go back to your code and resolve the problems.

Another aspect of mobile development that can add significant amounts of latency is network connectivity. While you cannot control the location of your users, or the network they are connected to, you can work to optimize the traffic that your app consumes, to ensure that the experience always runs smoothly. In Chapter 6, we covered the basics of network connectivity, tricks to simplify your data usage, and tools like Wireshark, MITMproxy, Fidler, and ARO to test that your connections are as optimized as possible. And finally, in Chapter 8, we looked at ways to get test results from your customers using real user monitoring (RUM) tools. By understanding where your customers' pain points are, you can work backward to ensure that their hurdles are removed, crashes resolved, and that your current (and future) users have a seamless experience in your app.

With the theories and tools outlined in this book, you should now have everything you need to poke, prod, and kick the tires of your Android app for performance gains. By digging into these tools and techniques, you will find ways to speed up the rendering and reduce the latency and battery drain of your app, which will ultimately improve performance.

Index

Symbols

3G networks, 177, 179, 203
4G (LTE) state machines, 176

A

ad-supported games, 29
Africa, 205
alarms, 35
Allocation Tracker, 140
Amazon
  effect of slow web pages, 3
  FireOS vs. Android SDK versions, 21
AMOLED (Active Matrix Organic LED), 32
Android application package (APK), 15
Android device lab
  Amazon devices, 21
  Android Open Source Project (AOSP) and, 20
  benefits of, 16
  cost of, 16
  CPU/memory and storage, 13
  determining representative device sample, 11
  device selection, 18-20
  dichotomy in device specifications, 12
  goals for, 16, 25
  infrastructure issues, 23
  networks in use, 13
  non-Google devices, 21
  Open Device Lab testing, 22
  OS versions in use, 12
  parameters tested, 17
  potential screen sizes, 12
  remote device testing, 22
  sample makeup of, 24
  SDK versions in use, 13
  variety of devices in use, 14
Android Emulator, 206
Android M
  App Standby feature, 47
  gfxinfo library, 101
  GPU rendering in, 100
Android Open Source Project (AOSP), 11, 20
Android Power profile, 30
Android Studio, 75, 102
Android Studio Monitor, 77
Android Wear, 20, 212
App Standby, 47
Application Quality Alliance (AQuA), 238
Application Resource Optimizer (ARO), 183-187, 196
Appurify, 22
ARM-based mobile chipsets, 157
ART runtime, 122, 124
asset reduction
  benefits of, 90
  Overdraw Avoidance system, 96
  overdraw in Hierarchy Viewer, 94
  overdraw testing, 91
  screen overdraw, 90
asset tinting, 90
AT&T Network Attenuator, 206
automatic sleep setting, 35

B

bandwidth
  defined, 174
  in 4G (LTE) connections, 177

Barnes & Noble, 21
Battery Historian, 52-62
Battery Historian 2.0, 62-66
battery life
  ad-supported games and, 29
  as primary complaint, xi
  battery drain analysis, 38-47
  battery drain causes, 29-38
  battery monitoring, 47-66
  improvement with Project Volta, 13
  improving by grouping connections, 198
  JobScheduler API, 66-70
  sensors affecting, 28, 34
  of smartphones, 8
  standby vs. in use times, 29
battery settings menus, 38
batterystats, 47-51
best practices
  basic performance rules, 188
  closing connections, 200
  detecting radio usage, 199
  file caching, 193
  file optimizations, 188
  goals of, 187
  grouping connections, 196
  image handling, 191
  regular repeated pings, 202
  security, 203
  for testing, 238
  text file minification, 190
big.LITTLE chipset design, 157
bitmaps, 126
Bluetooth, 212
brownouts, 6
bugs, 7

C

caching
  benefits of, 193
  best practices for, 196
  cache control, 194
  enabling, 193
  ETag response header, 194
  expires header, 195
  monitoring with Application Resource Optimizer, 196
  on the server, 194
cellular activity
  battery drain caused by, 33, 44, 175
  variety of networks in use, 174
  vs. Wi-Fi, 174
  worldwide coverage, 203-211
China, 13, 21, 204
circuit switched fallback, 179
compression, 192
Connectify, 180
connections
  closing, 200
  detecting, 199
  grouping, 196
  HTTP vs. HTTPS, 203
  regular repeated pings, 202
connectivity manager, 199
content delivery networks (CDNs), 204
cpuinfo command, 159
CPUs
  analyzing with Systrace, 160-163
  battery drain caused by, 33
  discovering blocking render, 113
  importance of optimizing, 157
  measuring usage, 158
  memory and storage issues, 13
  optimizing with Traceview, 163-169
  profiling with Trepn, 170
  task-based utilization of, 157
crashing
  Crashlytics SDK for, 216
  logfile notifications of, 218
  usage data, 225-230
Crashlytics SDK, 216
Crittercism, 227

D

Dalvik runtime, 122, 124
data usage
  analyzing with Fiddler, 181
  analyzing with MITMproxy, 183
  analyzing with Wireshark, 180
  battery drain caused by, 44
  closing connections, 200
  detecting connections, 199
  downloading data faster, 188
  during phone calls, 179
  grouping connections for, 196
  monitoring with Application Resource Optimizer, 183-187
  real user monitoring of, 225-230
  vs. radio connections, 178

(see also networks)
website testing, 187
Debug GPU Overdraw tool, 91
devices
average number of apps installed, 27
battery drain analysis, 38-47
battery performance in, 27
determining available memory, 126
determining representative sample of, 11
dichotomy in specifications in, 12
factors affecting OS breakdown, 12
non-Google, 21
obtaining older, 19
OS versions used, 12
potential number of, 1, 11
potential screen sizes, 12
rooted, 15
selecting for testing, 18-20
testing remotely, 22
top used Android, 19
variety in use, 14
wearable, 20
displays (see screens)
distributed denial-of-service (DDoS), 203
Doze framework, 37

E

Eclipse, 75
electromagnetic radiation, reducing, 206
ETag response headers, 194
Etsy
testing devices used by, 18
testing lab issues, 23
expires header, 194, 195

F

Facebook
battery drain caused by, 42
CDN mapping of, 205
data usage reduction by, 205
testing devices used by, 18
Faraday cage, 206
Fiddler, 181
file optimizations
caching, 193-196
downloading data faster, 188
images, 191
minification, 190
text file compression, 189
finite-state machines, 175
FireOS, 21
flexibly network aware (FNA), 207
Froyo, 12

G

garbage collection (GC), 123-126
GCM Network Manager APIs, 200
gfxinfo library, 101
Gingerbread, 12
Google Analytics, 226
Google Cloud Messenger, 202
Google cloud test service, 22
Google Glass, 20
Google Play, 4
GPS
battery drain caused by, 33
failover setting, 33
performance optimization, 211
GPU chips, 74
GPU rendering, 97-101, 117
Gzip compression, 189

H

hardware
Android features, 27
battery drain analysis, 38-47
battery drain causes, 29-38
battery monitoring, 47-66
with highest power drains, 31
Heap Dump, 138
Heisenberg, Werner, 35
Hierarchy Viewer, 77-89, 94
hourglass icons, 117
HTTPS traffic
decrypting with Fiddler, 181
decrypting with MITMproxy, 183
vs. HTTP, 203
hybrid apps, 187

I

Ice Cream Sandwich (ICS), 12
images
balancing size and quality of, 191
compressing, 192
downloading, 208
for various screen sizes, 191
grouping connections and, 196

- metadata in, 192
- thumbnails, 192
- WebP format, 193

India, 13, 21
Indonesia, 205
"insecure" builds, 15
Instagram, 90
instant gratification, 3
instant updates, 118
IP collisions, 202

J

jank
- analyzing for, 97-101
- decreasing/removing, 81-88
- defined, 74
- monitoring with Systrace tool, 103-116

Java runtime, 121
Jelly Bean (JB), 12
JobScheduler API, 66-70, 200
JPEG files, 126, 193
- (see also images)

Just in Time (JIT), 122

K

Keynote, 22
KitKat (KK), 12

L

last-mile latency, 211
latency
- accounting for, 210
- defined, 174
- in 3G networks, 177
- in 4G (LTE) state machines, 176
- in image downloads, 208
- in Wi-Fi connections, 174
- last-mile, 211
- measuring, 210
- network speeds compared, 174
- reducing with CDNs, 204
- state machines and, 176

LCD (liquid crystal display), 32
LeakCanary, 153
LED (light-emitting diode), 32
Lollipop, 12
lossy compression, 192
LTE networks, 13

M

man in the middle (MITM)
- Fiddler proxy, 181
- MITMproxy tool, 183

meminfo command, 126
memory, 13
- Android memory warnings, 136
- cleanup (garbage collection), 123-126
- determining app usage, 126-130
- dirty vs. clean, 122
- shared vs. private, 122

Memory Analyzer Tool (MAT), 145-152
memory leaks
- adding for testing, 142
- analyzing files in memory, 145
- analyzing with LeakCanary, 153
- analyzing with Memory Analyzer Tool, 145-152
- importance of diagnosing, 121
- memory management in Java, 137
- tracking with Allocation Tracker, 140
- tracking with Heap Dump, 138
- tracking with Procstats, 131-136

metadata, 192
minification, 190
minimum viable product (MVP) approach, 8
MITMproxy, 183
mobile app performance
- battery drain analysis, 41-44
- battery drain and, 8
- benefits of improving, 2-4, 10
- challenges of, 1
- defined, 1
- detecting performance issues, 9
- determining speed metrics, 238
- effect of CPU/memory and storage on, 13
- effects of poor, 4-9
- focus areas, 1
- measuring with Systrace tool, 103-116
- network optimization, 187-203
- opportunity for improvement, xi
- perceived performance, 117-119
- RCC (Radio Resource Control State) Machine optimization, 178

mobile apps
- 'future-proofing', 19
- average number installed, 27
- determining background memory usage, 131-136

- determining memory usage, 126
- loading times of, 73, 195
- measuring CPU usage, 158
- network-aware, 207
- primary complaint, xi
- scalability of, 4

Monitor, 102, 138
MVP (minimum viable product) approach, 8

N

negative reviews, 7
nesting behavior, viewing, 77
Network Activity Sample app, 207
Network Attenuator, 206
network-aware apps, 207
networks
- 3G, 177
- Bluetooth, 212
- data usage during phone calls, 179
- global differences in, 13, 173
- global market penetration by "G", 203
- GPS, 211
- grouping connections to, 196
- mobile data traffic analysis, 179-187
- network speeds compared, 174
- optimizing for Android, 187-203
- orphaned connections to, 178
- radio vs. data connections, 178
- RCC (Radio Resource Control State) Machines, 176-179
- testing on slow, 205
- variety in use, 14
- WiFi vs. cellular connections, 174
- worldwide cellular coverage, 203-211

Nexus 6, 12
Nexus devices, 19
Nokia X AOSP, 21
Nook tablet, 21

O

onTrimMemory command, 136
Open Device Lab, 22
OpenWRT, 206
optimistic action, 118
organizational performance
- determining speed metrics, 238
- development issues, 237
- getting buy-in, 235
- importance of, 235
- motivating teams, 236
- testing tips, 238

orphaned connections, 178
OS versions
- currently in use, 12
- garbage collection in, 123
- testing latest, 19

outages
- costs of, 4
- rolling, 6

overdraw
- benefits and drawbacks of, 90
- Overdraw Avoidance system, 96
- testing, 91
- visualizing in Debug GPU, 92
- visualizing in Hierarchy Viewer, 94

Overdraw counter, 91

P

page loading times, 73, 118, 195
page refresh rates, 74
perceived performance, 117-119
Perfecto Mobile, 22
performance bugs, 7
performance improvement
- benefits of, xi, 2-4
- detecting performance issues, 9
- feasibility of code optimization, 1
- focus areas, 1
- primary focus of, 6

performance metrics, 238
PhoneGap, 187
pixel density, 12
PNG files, 126
power management
- ad-supported games, 29
- Android Power profile, 30
- battery drain analysis, 38-47
- battery drain causes, 29-38
- battery monitoring, 47-66
- Doze framework for, 37
- importance of, 27
- improvement with Project Volta, 13
- improving by grouping connections, 198
- sensors affecting, 28, 34
- smartphone battery drain, 8

PowerManager API, 36
Procstats, 131-136
Profile GPU Rendering, 97-101

progress bars, 117
Project Butter, 13, 75, 103
Project Svelte, 13
Project Volta, 13
proportional set size (PSS), 127

R

radios
- battery drain caused by, 33
- Bluetooth, 212
- detecting usage, 199
- GPS, 211
- radio vs. data connections, 178
  - (see also networks)

rankings, 4
RCC (Radio Resource Control State) Machines
- 4G (LTE) state machines, 176
- benefits of, 176
- defined, 175
- drawbacks of, 176
- grouping connections for, 197
- optimizing apps for, 178
- radio vs. data connections, 178

real user monitoring (RUM), 10
- availability of, 216
- benefits of, 215, 231
- crashing, 218-230
- enabling tools for, 216
- sample app, 217
- SDK performance, 231

refresh rates, 74
retina display, 191
rolling outages, 6
root access, 15
RTTs (round trip times), 209
RUM (see real user monitoring)

S

Samsung Galaxy S, 12
Samsung S3, 13, 19
Samsung S5, 28
screens
- battery drain caused by, 32
- optimizing image files by size of, 191
- overdrawing, 90-96
- potential sizes of, 12
- refresh rates, 74
- Systrace screen painting, 106-113

SDK versions
- Fire OS vs. Android versions, 21
- grouping connections and, 196
- variations in, 13

search engine rankings, 4
sensors
- accessing battery drain caused by, 34
- included in Samsung S5, 28

skipped frames
- discovering with Systrace, 102-116
- due to CPU blocking, 113

sleep settings, 35
smartphones
- Android penetration, 11
- battery life of, 8
- battery performance in, 27
- lifespan of, 13
- market share of, 203
- screen sizes of, 12

smartwatches, 20
Souder's performance rules, 188
South Africa, 13
spinners, 117
Spotify
- battery drain caused by, 42

Square, 153
state machines, 175
storage, 13
Super LCD3 screens, 32
superusers, 15
synthetic testing, 10
Systrace
- 2015 update, 115
- benefits of, 103
- best application of, 103
- CPU analysis with, 160-163
- CPU usage blocking render, 113
- evolution of, 105
- screen painting, 106-113
- starting with, 104

T

Testdroid, 22
testing
- Amazon devices, 21
- Android Open Source Project (AOSP) and, 20
- best practices for, 238
- building device labs, 15-24
- CPU/memory and storage, 13

determining representative device sample, 11
goals for, 16, 25
infrastructure issues, 23
networks in use, 13
non-Google devices, 21
Open Device Lab, 22
OS versions in use, 12
potential screen sizes, 12
real user monitoring, 10
remote device testing, 22
sample device lab makeup, 24
SDK versions in use, 13
synthetic, 10
variety of devices in use, 14
text files
compressing, 189
minification of, 190
thumbnail images, 192
top command, 158
Traceview
Android Studio, 166-169
Legacy Monitor DDMS tool, 163-166
Trepn, 170

U

uncertainty principle, 35
user experience
effect of outages on, 5
effect of performance bugs, 7
importance of speed to, 2, 73, 195
improving worldwide use, 203-211
perceived performance and, 117-119
real user monitoring of, 225-230
recent Android improvements, 74
screen size and, 12
slow vs. crashing apps, 6
top frustrations, 8
user interface (UI)
analyzing for jank, 97-101
asset reduction, 90-96
building views, 75-89
page refresh rate, 74
perceived performance and, 117-119
performance benchmarks, 73
performance issues, 73
recent Android improvements, 74
skipped frames, 102-117
USS (Unique Set Size), 127

V

views
design view of app layout, 75
finding redundant, 77
Hierarchy Viewer tool, 77-89
remeasuring, 76
rendering steps, 76
reusing, 89
Voice over LTE (VoLTE), 179
VSYNC buffering, 75, 106

W

wakelocks, 35
finding bad, 61
Walmart
effect of slow web pages, 3
wearable devices, 20
web performance, xii
cost of slow websites, 3
cost of website outages, 4
testing tools, 187
WebP image format, 193
WebPageTest, 187
Wi-Fi
bandwidth and latency in, 174
battery drain caused by, 33, 174
emulating with Connectify, 180
security issues, 203
testing with Wi-Fi throttling, 206
vs. cellular radios, 174
Wireshark, 180, 212

Y

YouMail, 4

Z

Zygote process, 122

About the Author

Doug Sillars is the performance outreach lead at the AT&T Developer Program. He has helped thousands of mobile developers apply performance best practices to apps. The tools and best practices he has developed at AT&T help developers make mobile apps run faster by using less data and less battery. He and his wife live on an island in Washington State, where they homeschool their three children.

Colophon

The animal on the cover of *High Performance Android Apps* is a *brown noddy (Anous stolidus)*. It is part of the gulls and terns family and can be found near warm, tropical waters. This bird is also known as a common noddy.

The brown noddy has a distinct look. The feathers are brown in color (hence its name) except for on the forehead, which are whitish-gray in coloring and extend over the tops of the eyes. The tail is wedge-shaped and of a darker brown coloring than the majority of the body. The tips of the wings are also of this darker color. The brown noddy's legs, feet, and bill are also blackish-brown. Males and females look identical, but females are smaller in size.

Like most seabirds, the brown noddy's diet relies heavily on what the tropical waters and oceans they live around can provide. They hunt for small fish from above the surface, but also collect the leftovers of prey that rise up from underwater predators. These birds will also forage for meals along the shore and in lagoons.

Breeding and nesting grounds for the brown noddy are fairly diverse, but always take place inshore. They will build nests in trees, shrubs, cliffs, and on beaches and docks. They only lay one egg per year, which both parents will take turns incubating. This parental partnership continues even after the egg hatches, until the chick leaves the nest around eight weeks of age.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from *British Birds*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.