O'REILLY®

High Performance Android Apps

IMPROVE RATINGS WITH SPEED, OPTIMIZATIONS, AND TESTING

Early Release

RAW & UNEDITED

Doug Sillars

High Performance Android Apps

Doug Sillars

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

High Performance Android Apps

by Doug Sillars

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*.

Editor: Brian Anderson
Production Editor: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!

Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition:

2014-11-03: Early release revision 1

2015-01-03: Early release revision 2

2015-03-03: Early release revision 3

2015-05-04: Early release revision 4

2015-07-20: Early release revision 5

See *http://oreilly.com/catalog/errata.csp?isbn=9781491912485* for release details.

ISBN: 978-1-491-91248-5

[?]

Table of Contents

Preface

You are building an Android application (or you already have.) Despite this, are you not totally happy with your apps 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 potential performance improvements. The goal of this book is to help understand the pitfalls of mobile performance, expose some of the tools to test for issues so that you can catch any major performance issues in your mobile application before it impacts your customers.

Studies have shown that customers *expect* mobile applications to load quickly, to rapidly respond to user interactions, and be smooth and pleasing to the eye. As application get faster, user engagement and revenues increase. Mobile applications 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 #1 complaint to carriers and device manufacturers is 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 were having.

The consensus is clear: mobile applications that are fast and run smoothly are used more often and make more money for the 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 application.

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 application performance. Developers of mobile applications 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 where developers share tips on how to *make the web fast*. Steve Souders wrote "High Performance Web Sites" in 2007, and the topic is covered in books, blogs, conferences, etc.

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 OSes have matured, customers are realizing that mobile apps have a role in the performance of their phone.

There are many great tools for measuring Android app performance, but until now - no guide that lists 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 application get faster while not killing your customer's battery.

Navigating This Book

- Chapter 1 is an introduction to mobile app performance. We'll run the numbers to show how crucial performance is to your application. I'll highlight many of the challenges, but also the effects of poor performance in the marketplace. These are the stats that you can use to convince your management that spending time speeding up your applications is time well spent. tTe data presented here generally holds for all mobile platforms and devices.
- Chapter 2 covers testing. Android is a huge ecosystem with tens of thousands of devices with different UIs, screens, processors OS versions etc. 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 is all about the battery. What causes drain, and by how much. How your customers may discover battery issues in your app, and developer tools to isolate battery issues. I also cover the new JobScheduler API (released in Lollipop) that abstracts application wakeups to the OS.
- Chapter 4 Screen Perf. Being the largest power drain on your phone, and the primary interface to your app - this is where slow apps show jank (skipped frames) and slow rendering. We'll walk 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.

- Chapters 5 and 6 look at Memory and CPU issues like garbage collection and memory leaks and how they affect the performance of your app. Testing with tools like Procstats, MAT and TraceView - you'll learn to dig into your app to discover potential issues.
- Chapter 7 looks at the network performance of your app. This is where I got started in mobile perf, and we'll look into the black box of how your app is communicating with your servers and how we might optimize these communications. We'll also look at how to test the performance of your application on slower networks (since much of the developing world will be on 2G and 3G for decades to come.)
- Chapter 8 Using Real User Measurements 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 customer's devices.
- Appendix Covers organizational performance. How to get buy-in to building 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.

Online Resources

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

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.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

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: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

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"permissions@oreilly.com*.

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 product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers 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 dozens 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/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.combookquestions@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, Carolyn). I would like to especially shout out to the ARO team: Jen, Bill, Lucinda, Tiffany Pete, Savitha, John and Rod (and of course all the devs!) who work everyday 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 application performance.

A big thank you to all of those who read the early iterations of the book - thank you for your comments tips and suggestions. 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

Its 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 application works, how efficiently it works, or if the app 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 application 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 be looking at app performance specifically in terms of an application's 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 applications. 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 applications. Performance leads, single developers, 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 application. This may not always be feasible, but knowing where your apps 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 application, you will be ready to profile when you feel like there is an issue. Knowing the tricks to improve the efficiency, performance and the speed of your application 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 100ms are perceived as instant, where actions that take a second or more allow the human mind to become distracted. Delays and slowness in your app (even if just the *perception* of slowness) is probably one of the biggest killer of app engagement out there. It can also potentially damage your customer's phones too! (a study in 2012 found that slow apps caused 4% of users to throw their phone!).

E-Commerce and Performance

Imagine an e-commerce application. This application has collected analytics showing that the average e-commerce session is 5 minutes long, and each screen load takes an average of 10 seconds to complete. Your screen view budget/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.

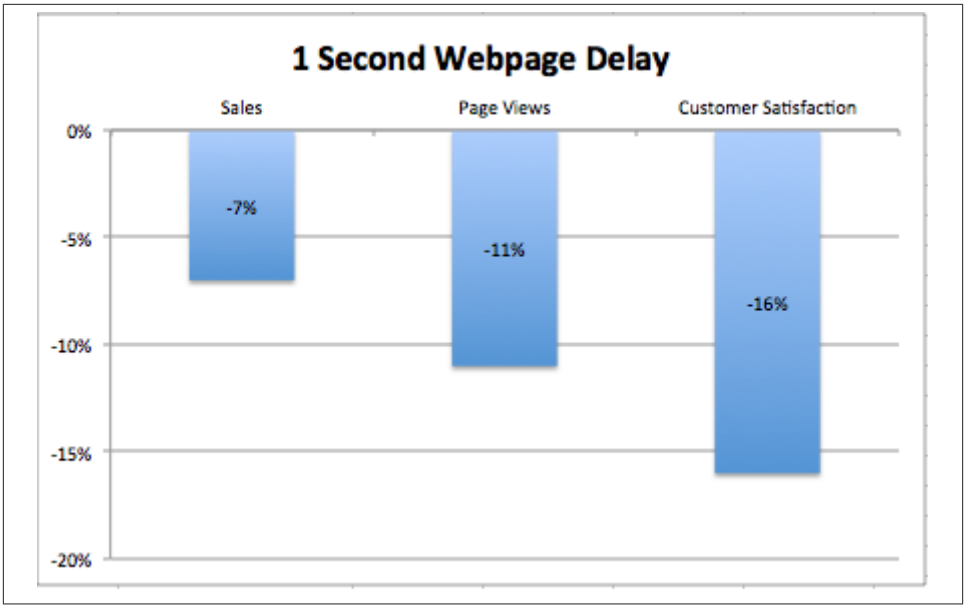


Figure 1-1. Effects of Slow Websites

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

Performance studies on web performance provide a lot of context to mobile application 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 and Walmart have independently reported similar statistics. Both major retailers found that just 100ms of delay to on their desktop webpages caused their revenue dropped by 1%. Shopzilla rearchitected their website for performance, and saw page views increase by 25%, increased conversions by 7-12%, and actually used half the nodes they previously required!

Beyond e-commerce Sales

In addition to decreased sales and revenue, mobile applications with poor performance get lower rankings in Google Play. Even worse are the stories of badly behaved apps being pulled from customer devices. In 2011, T-Mobile asked Google remove the YouMail application from the Android Market. YouMail is a 3rd party voicemail app, and the way the application 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 con-

nection 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 applications 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 applications 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 application, 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 four studies[1] *http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html* pass:[*http://info.appdynamics.com/DC-Report-DevOps-and-the-Cost-of-Downtime.html*] estimating the costs of an outage (and they are rising). Two of these 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 outage causes a $175k loss in revenue per hour in 2015, and the costs are just getting higher:

1. *http://www.datacenterdynamics.com/critical-environment/one-minute-of-data-center-downtime-costs-us7900-on-average/83956.fullarticle*

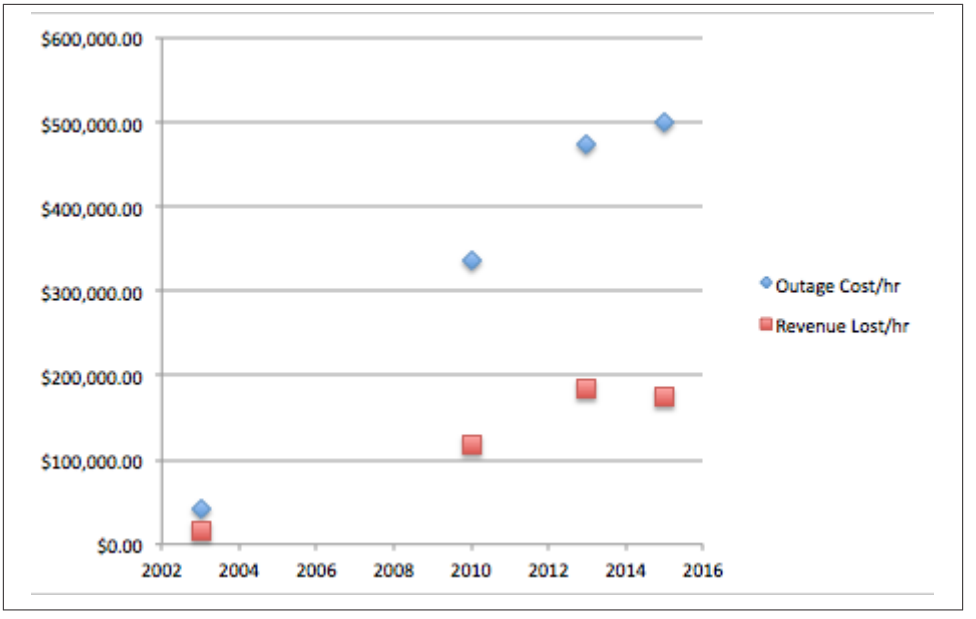


Figure 1-2. Cost of an Outage per hour

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. We know that there is loss in revenue, customer satisfaction, and when there is an outage, our customers have no idea how to react:

Figure 1-3. Facebook Users' Reaction During Outage

But, in all seriousness, uptime performance is crucial to the survival of your company. The mobile analogy to an outage is when your application crashes. Obviously, the first performance issue you must resolve are crashes, since if the app doesn't work, it doesn't matter how fast it is. However, When your application 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 an electricity brownout, your utility is providing a lower voltage, and your lights appear dim (and your fridge might stop working all together). A slow Android app operates in the same way. Your customers can still use your application, but scrolling may be laggy, images may be slow to load and the whole experience *feels* slow. Just as a brownout adversely affects electricity users, a slow android app is equivalent to a rolling ongoing outage. A study released by HP in March 2015 shows that customer's have the same reactions to slow apps that they have to apps that crash:

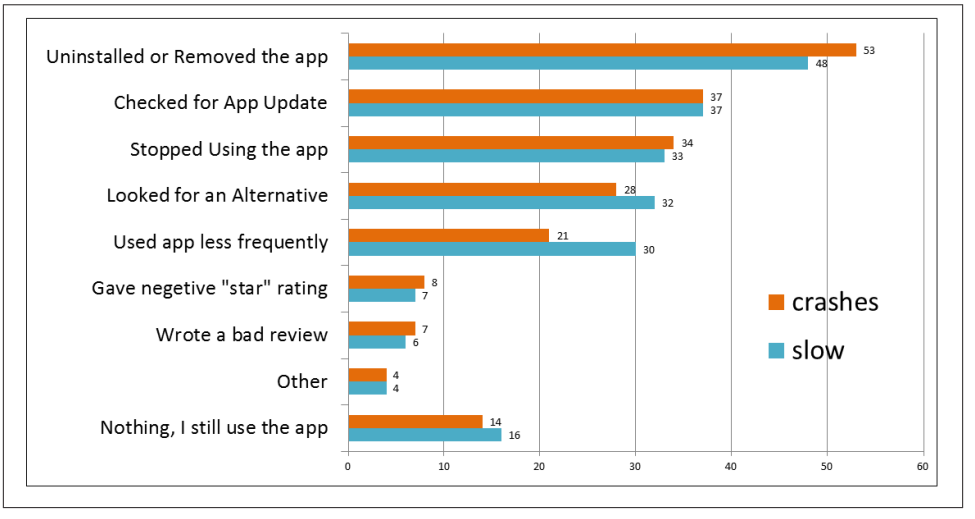


Figure 1-4. Poor Performance vs. Crashes: Same Consumer Result

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

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

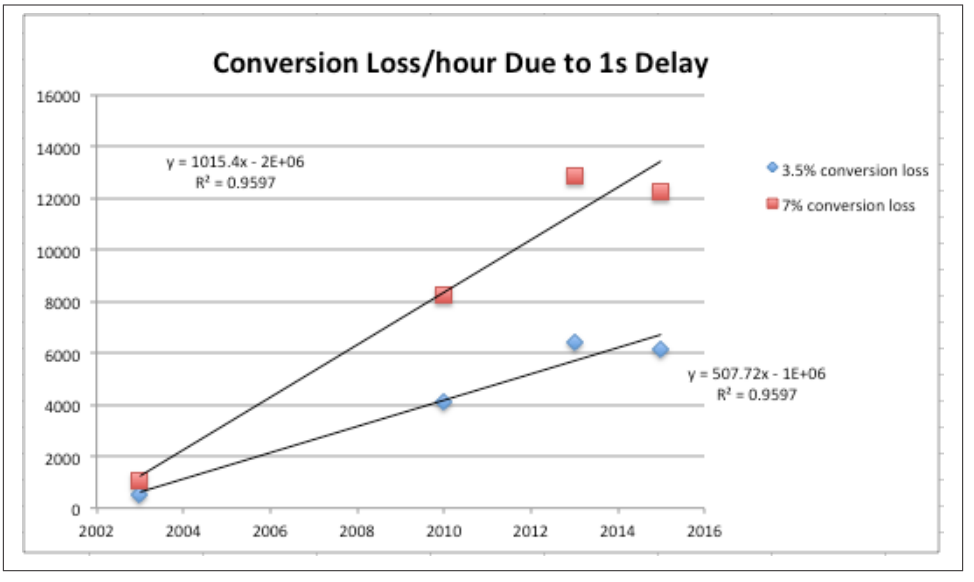


Figure 1-5. Cost of a Slow App Per Hour

As the graph shows, the cost of a slow app is increasing year over year. As your application 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 passed on to the developers by users leaving feedback in Google Play reviews. 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 discourages download of your app is costing you money!

Once the application has been downloaded, you are not out of the woods. In 2014, 16% of downloaded Android apps are launched only once. Customers are easily distracted, and with so many choices in the app markets, if your app doesn't satisfy, they will quickly try a similar app. Now there could be many reasons that users abandon apps. It can be argued that being frustrated with an application is a top reason to abandon or uninstall. According to a study by Perfecto Mobile the top user frustrations are:

1. User Interface Issues (58%)
2. Performance (52%)
3. Functionality (50%)
4. Device Compatibility (45%)

While performance is directly called out as the #2 reason for customer frustration, it is clear that the other top 4 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 Minimal Viable Product (or MVP) approach to your Android app, where the initial launch contains bugs and and performance sinks assumes that when the fixes are made, you:

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

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

a. Actually updated
b. Opened up so that the fixes are seen.

Smartphone Battery Life - The Canary in the Coal Mine

The studies above 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, applications (and especially non-optimized applications) can be a MAJOR factor in battery drain. I use my end of 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 the reviewer set up the new device with the same applications that were on the previous device, it will be similar - there is a rogue app installed on both devices that 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 perfor-

mance, and how improving the performance of your app will extend your customers battery life.

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 device, the device manufacturer or the carrier for device battery issues. This is no longer the case. In fact, a 35% of consumers have uninstalled an app due to excessive battery drain. The tools available to consumers displaying how applications drain the battery only just now coming to market, but the quality of these tools are radially 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 battery and power conscious as you can while architecting and building your mobile applications.

Testing Your App For Performance Issues

The best way (pre-launch) 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 bak 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 RUM (Real User Measurements).

Synthetic Testing

Synthetic tests are created in the lab, to test specific use cases, or perhaps to mimic user behaviors in your mobile application. 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 work many bugs and performance issues. However, with 19.000 Android User agents reported by Akamai, there is no way you can possibly run a synthetic test for every possible scenario.

Real User Measurements (RUM) Testing

Since testing every device scenario is impossible in the lab, it is essential to collect real performance data from your customers. By inserting analytics libraries into your application, you can collect realtime data from all of your users - allowing you to quickly

understand they types of issues they might be facing. This gives you the chance to respond to customer issues/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 YY will walk through some typical results you obtain from RUM measurement tools.

Conclusion

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

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 is 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 seen each day. 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 US 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 that (without automation) that you will be regularly testing on this many devices. In this chapter, we'll walk through a few different options to building out an Android device lab that will help you maximize your UI, functional and performance testing, on a minimum of devices.

What Devices Are your Customers Using?

The easiest way to break down Android usage is by version of OS - since your performance tuning will vary by the OS in use. As of April 2015, 5.5% of Android users are running a version of Lollipop, only 41.4% are running KitKat (40.7% on Jelly Bean, 5.7% on ICS, and 6.8% of devices on Gingerbread and Froyo - the only levels currently re-

ported by Google). 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 3 years ago. The device breakdown will also vary depending on the region of the world, as higher end device sales will predominate in wealthy countries, and lower powered new devices and resale of used devices predominate in developing countries.

Device Resale
Did you remember to factory reset your old devices before selling them on Craigslist?

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 rear facing) and some have cameras front and back. Start throwing in sensors like NFC, Thermometer, accelerometer 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 US seem to be getting larger and larger, with no regard for hand (or pocket) size. The Samsung Galaxy S (2010) was 122 mm high, and featured a pixel density of 480x800, whereas the S5 is 145 mm high, and 1080x1920 (nearly an inch longer, and 5.4x the pixel density in just 4 years!). The latest Nexus 6 from Motorola maxes out the phablet category of devices with a massive 151mm (that's 5.92 inch) high screen that is 2560x1440 Quad HD resolution.

Despite this trend to larger and larger screens, there is still a sizable chunk of users with screens of 480x320 or smaller (>5% in Q2 2014) - and 17% of users in South Africa are using phones 240x320, 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. Devices post Jelly Bean 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 2 years. The Samsung S3 (released in 2012) is still being sold as a new device in 2015 (alongside is predecessors 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 2 phones in India are 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 US and Europe are common, as are single core devices running Gingerbread. Your device may run smoothly on common devices in the US, but how does it react to a lower powered device with less RAM or storage. As your application 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 (latest studies show 97% of the population has access to LTE or other 4G technologies.) This drops to 8x% in Western Europe, and drops from there 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 Device is Not Your Customer's Device

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 that developers are carrying devices equivalent to the Nexus 6, Samsung S6 or HTC One (M9) - with 4 or 8 CPUs, multi-Megapixel cameras, 16-32 GB of memory, 2-3 GB of RAM, HD video recording, running Lollipop. 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 the internet. These users will be looking at low cost devices for access, 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 2-3 years ago that you loaned to a family member and forgot about.) These devices not only lack the horsepower we take for granted - but we also have to realize the other limitations these users may face - access to electricity to charge their phones, and the quality of the mobile network that supplies the data.

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 application 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 can access any file- even in your protected sandbox, meaning that anything sensitive cannot be stored on the device.)

Rooted ROMs typically have a superuser apk installed as an interface between applications 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 application on the device. This is a very powerful option, and a useful one for Android developers. On the downside, many Android applications 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. It is currently legal to root an Android phones in the US (but not tablets), as long as they are no copyright infringements. If you are unclear on the legality in your jurisdiction, consult with legal counsel.

Testing

So, with the challenges described above, 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 you are equally supporting your 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.

Here are a few approaches towards building a device lab that covers your bases today and will keep you covered moving forward. Of course, everyone'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 application 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 application in market, your analytics data can be very helpful to break

down the devices that your customers are using. (For more information on application analytics, jump to Chapter 8.) Perhaps your user's device choices deviate from the top reported devices. To keep existing customers happy, you should ensure you always test on customer's top devices. If you don't have analytics specific to your application, 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 pretty safe bet).

You Want $X,000 for Devices?

Cost is always is the elephant in the room. In this section 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, inability to use sensors like location and accelerometer to name a few.) You'll have to convince your management that device 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, lets take a look at a variety of parameters or tests you may want to iterate over with different devices.

1. Screen Size
    a. small (4.4%)
    b. normal (82.9%)
    c. "phablet" (8.6%)
    d. Tablet (4.1%)
    e. Special cases (Wear, TV, Auto)
2. Screen Density
    a. Low (4.8%)
    b. Medium (16.1%)
    c. High (40.2%)
    d. Extra-High (36.6%)
3. Processor
    a. Dual Core
    b. Quad Core
    c. Octo Core

4. Memory
    a. RAM
    b. Storage (perhaps devices with nearly full memory vs. devices with lots of space)
5. Network speed
    a. 3G
    b. LTE
    c. Wi-Fi
6. SDK version
    a. Gingerbread (2.3 vs. 2.3.3 vs. 2.3.7)
    b. Ice Bream Sandwich
    c. Jelly Bean (4.1 vs. 4.3)
    d. Kitkat
    e. Lollipop
    f. M (and beyond)
7. Etc.
    a. Rooted
    b. security testing
    c. 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 a hundred 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. For example:

Facebook has gone an interesting route for their device testing. 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 them 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 their "2011" phone class, a dual core, 1 GB RAM device (and they use the Samsung S2 to test this class of devices).

Etsy uses their device analytics to discover which devices are popular, and did their initial sourcing from that list. They source used devices that do not have mint batteries so they may test devices that have more realistic power drain for older handsets. As new devices are released, they watch to discover which devices growing quickly in their user base.

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 different areas, so looking at older devices on earlier SDKs without there 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 to: . Save money . 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 480x800, so perhaps choosing an older device with a smaller screen (to maximize your device portfolio). Perhaps a device like the Samsung Galaxy Y (with a 240x320 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 several online sources (2014) 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 Lollipop, Google began pre-releasing the latest OS to all developers on the Nexus 5 and 7, 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, TV and Auto. These platforms are different form traditional Android, but depending on your development plans, you may want to integrate some of these devices 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 smart watches that communicate back to an Android device over Bluetooth (there is no unique phone number or SIM on the device). Like with Glass, 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:

1. Suggestions: a timely list of information for the user like messages, location relevant data, etc.
2. 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 applications 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 US when discussing Android is that Android is 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 focused completely on the various Google incarnations of Android on purpose, as they are the predominate devices in the US. However, in an effort to be complete in my coverage of Android, let's take a look at other common forks of the AOSP.

Recent studies (Summer 2014) 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:

1. Google Play Store for app distribution
2. Google Cloud messenger push notification

3. Google Play Services
4. Google products and apps amongst other tools etc 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 devices in the United States running a fork of Android are Amazon's Popular e-readers, the Kindle. Amazon has also launched into phones with the Amazon Fire Phone, and TV set top boxes with FireTV. Amazon calls its fork of Android FireOS, 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

It might be useful to have a few Amazon devices in your lab, as Amazon does have its own unique appstore to deliver content to all of the Fire tablets - this is another market for your Android applications. 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 appstore are also available on Blackberry devices which has a runtime that allows for Android apps ti 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 the Barnes & Noble Nook tablets. Nokia had a short-lived Nokia X AOSP project before being acquired by Microsoft.

Outside of the US, 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 1B 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

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, 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 applications. These services typically have many top phones, and allow you to run scripts, or other CI 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 he physical devices, you are unlikely to see slowness or performance issues - you'll 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 and virtual devices in "every language, orientation and network condition." 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. These are grass roots device labs (some have permanent homes, some do not) of devices that are available for testing. 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.

Figure 2-1. Worldwide Open Device Lab Locations

Other 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. 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, are not accidentally upgraded
- Having the appropriate cables and chargers for each device.

These additional details a crucial to get 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.

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 - controlling your Android coffeemaker from bed, or etc.)

Making sure that your application runs well on phones, tablets, cars, TVs, watches, sunglasses, etc. 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 application, 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 application 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 in our bags, we have chargers at home, work and in our car 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. 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 application. 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 device, it seems that there is nothing you cannot do with the device. However, as Uncle Ben told young Peter Parker (the fledgling Spider-Man) "With great power, comes great responsibility." Android devices 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 aspects that might give your customers alarm.

If you look at the sensors included in the Samsung S5, there it is pretty amazing the power we have in our hand today:

1. Fingerprint Scanner
2. Heart Rate Monitor
3. Light monitor
4. Relative Humidity
5. Environment Temperature
6. Barometer
7. NFC
8. Gyroscope
9. Accelerometer
10. Bluetooth
11. Wi-Fi
12. FM Radio
13. Cellular radio
14. Front and Back Camera
15. GPS
16. Magnetic Field
17. Light Flux
18. Battery Temperature
19. Microphone
20. Touch

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 customer's Android device, 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.

Nowhere is this more evident than downloading an Android free ad-supported causal game. Sometimes, after playing these games for a 10-15 minutes of playing, 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 etc. All of these components working at once drains the battery so quickly that it heats up. A study released in March 2015 found that applications with ads used 56% more CPU, 22% more memory 15% more battery over the same app with ads stripped out.

It is my contention that most battery issues with mobile devices is not hardware related, but poorly designed applications that misuse the capabilities of the device. In this chapter we'll walk through some of the mis-steps of hardware usage, and how to avoid them in your Android application (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 applications that are using excessive battery. By learning these techniques, you'll discover if your customers will discover similar performance issues with *your* app on their phone. By understanding how Android grades applications 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.)

Android Power Profile

As we'll discuss later in the chapter, the battery settings menu reports the percentage of battery drain for each application 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 application 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 Profile XML.

```
<?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 application, you'll also be improving the battery life of your user's 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 to your local machine. You'll need to decompile the apk to extract the res/xml/power_profile.xml file. The (to come) section of [Link to Come] walks through the decompilation steps.

All of the values in the Power Profile are reported in milliAmps (mA). As you recall from Physics, 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 mAh - milliamp-hours, or the amount of current that flows in one hour.

Screen

As seen in the Power Profile XML, 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 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 application 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 2 major screen types in Android devices, LED (Light Emitting Diode) and LCD (Liquid Crystal Display). Manufacturers have proprietray versions of these screens (e.g. AMOLED - an Active Matrix Organic LED, 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

LCD screens (in simple terms) 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. Since each pixel is represented by 3 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, while 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.

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

Radios

As the Power Profile XML 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 application'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 coarse location, the cellular network can often provide enough data that the GPS radio might never turn on. Since 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 application 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 application is a game or has a lot of heavy calculations, you know you will be hammering the CPU hard. Additionally, if your application 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 applications 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 make sense to your application. 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 form the Monsoon into them.)

Get To Sleep!

Letting your app go to sleep when it is not doing anything is important. Releasing the sensors, radios and allowing the screen to go 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 application wakes up the device, you will go a long way to saving your customers' battery.

Wakelocks and Alarms

Historically, developers have used wakelocks and alarms to wake up a device to process information. Since it is likely that you might want your application to 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 applications to process events, perhaps turn on the radio etc. In Lollipop, Android added a new "JobScheduler" on page 62 API, which allows for smarter and synched device wake ups.

Wakelocks

Wakelocks give you the ability to wake up (or keep awake) parts of the mobile device. This is a crucial feature in applications when used properly. 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 Detector applications in Google Play that are useful for diagnosing Wakelock issues. These applications all generally do the same analysis, so pick one to test your applicaion'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 application using these tools.

Alarms

Alarms allow you to set the time that specific operations will be run. These are typically run when your application 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 applications that used alarms to sync data. This app turned on the customer's phone and cellular radio 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 application). Otherwise, you can use an inexact alarm where the OS will coordinate all of the alarms to minimize battery drain. The example below 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 application uses will accellerate the drain. Studies have shown that 70% of battery drain when the device is idle is caused by applications 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 upcoming Android M release, 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 applications, 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 applications 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 with Android M 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 application 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 applications 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 application not appear as a battery hog in these menus, as it is a great tipoff to your customers to uninstall your app.

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 applications that have contributed to battery drain over the selected period. Let's walk through what these graphs tell you (and your customers).

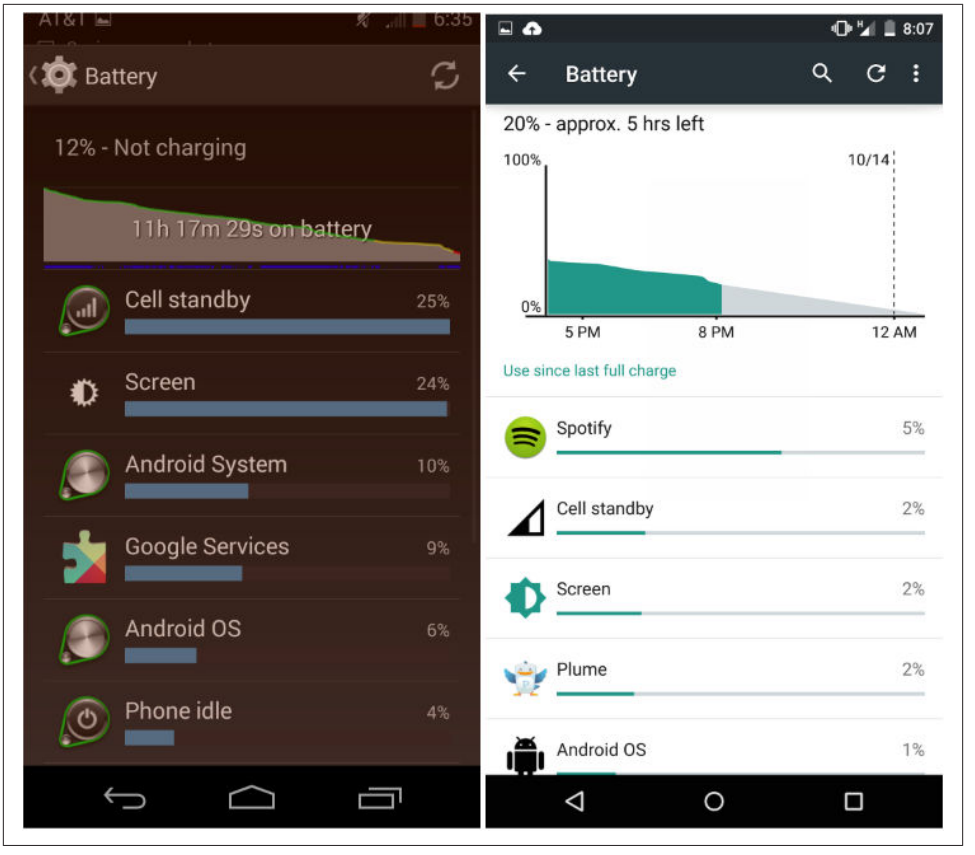


Figure 3-1. The KitKat (left) and Lollipop (right) Battery Menus

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, the graph expands and shows more device specific details about what the device has been doing.

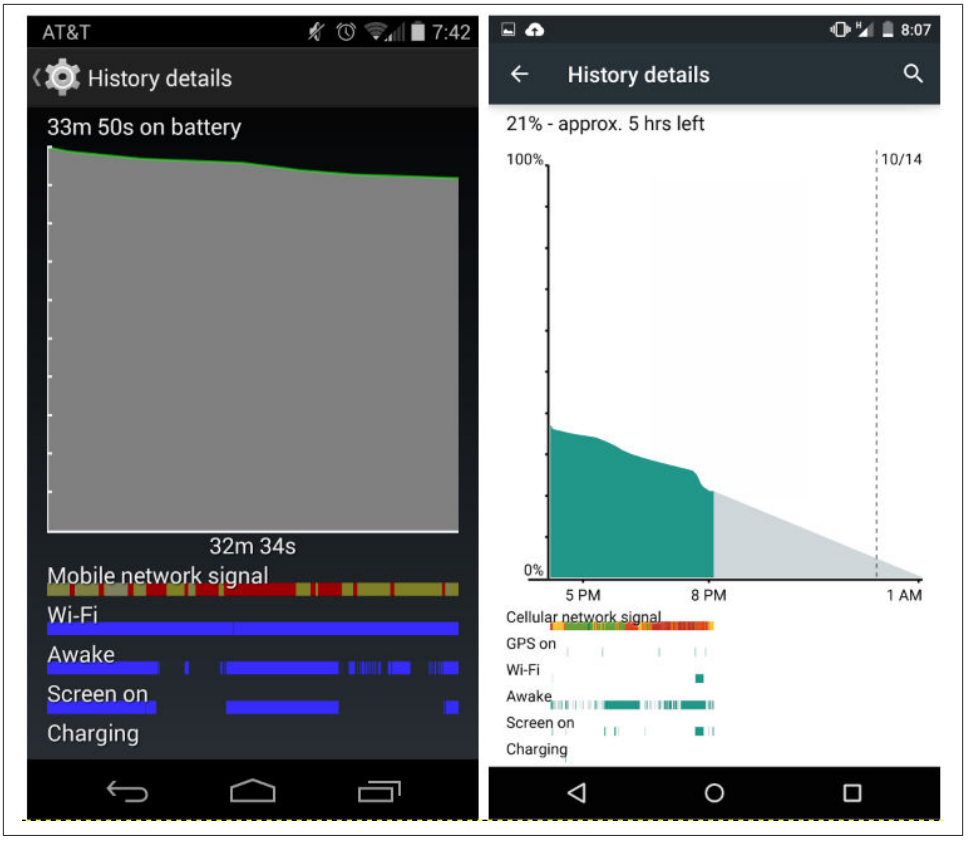


Figure 3-2. KitKat (L) and Lollipop ® Battery Details

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 application performance, I prefer the KitKat view, since 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 application using a wakelock or alarm to use the device while you (the customer) is not using

it. If you see this occurring frequently, you can look at the applications 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 application associated with battery drain. In my experience, the percentages are never very large, but this might be because every application is responsible for a very small percentage of battery drain. By selecting a specific application from the menu, You'll see you the CPU usage of your app in the foreground and total. 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 (yeah! 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, here are menus from Facebook and Spotify from the KitKat Battery Menu:

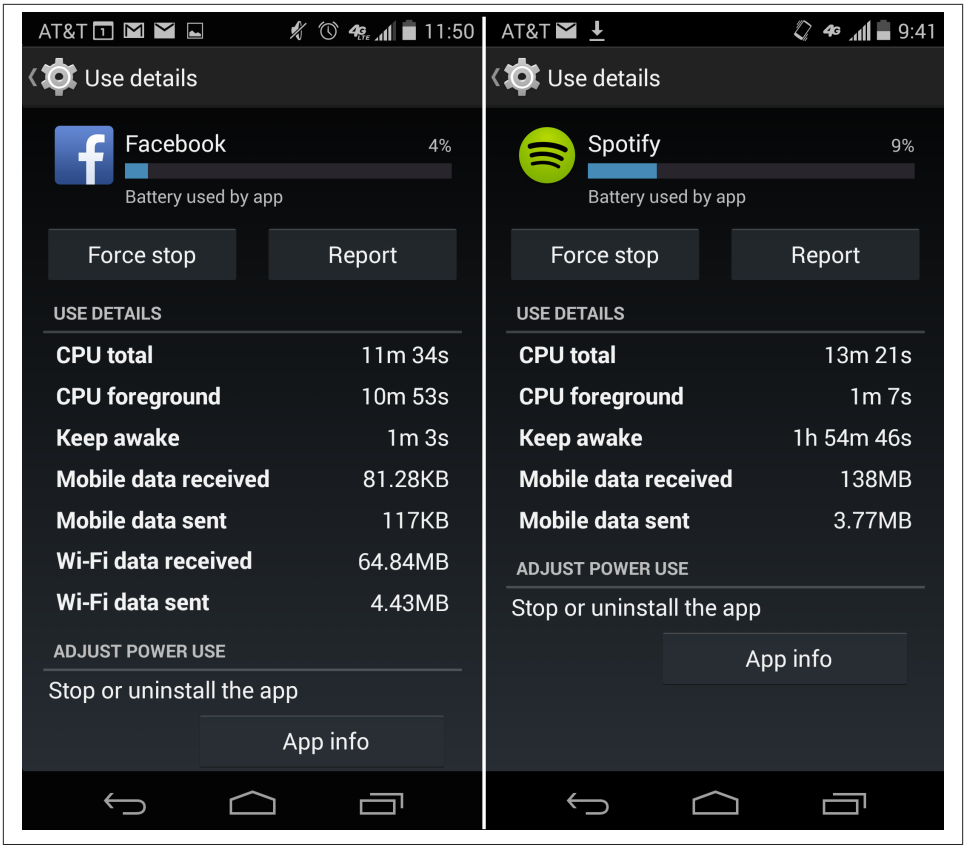


Figure 3-3. Facebook and Spotify Battery details

In this view, Facebook is being credited with (blamed for?) 4% of the battery drain (calculated with "Android Power Profile" on page 27) on the device. Facebook's CPU usage is primarily in the foreground (11 minutes out of 11.5 minutes). The addition 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 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 data traffic is hight,

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:

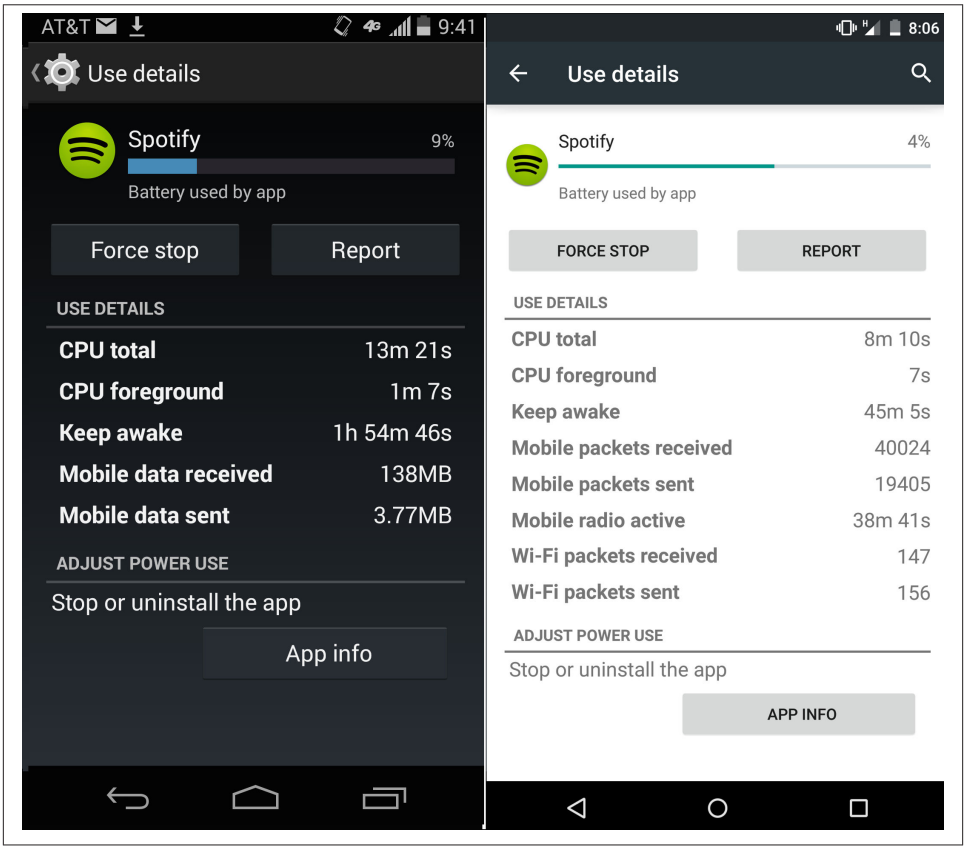


Figure 3-4. Spotify Battery details in KitKat and Lollipop

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 application is also reported. These are powerful new reports, since 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 application, you are provided with the amount of data used in the foreground and background. In this case, I have moved the sliders to show only the data for 2 days, allowing my to pinpoint foreground and background data usage for Facebook for just 24 hours.

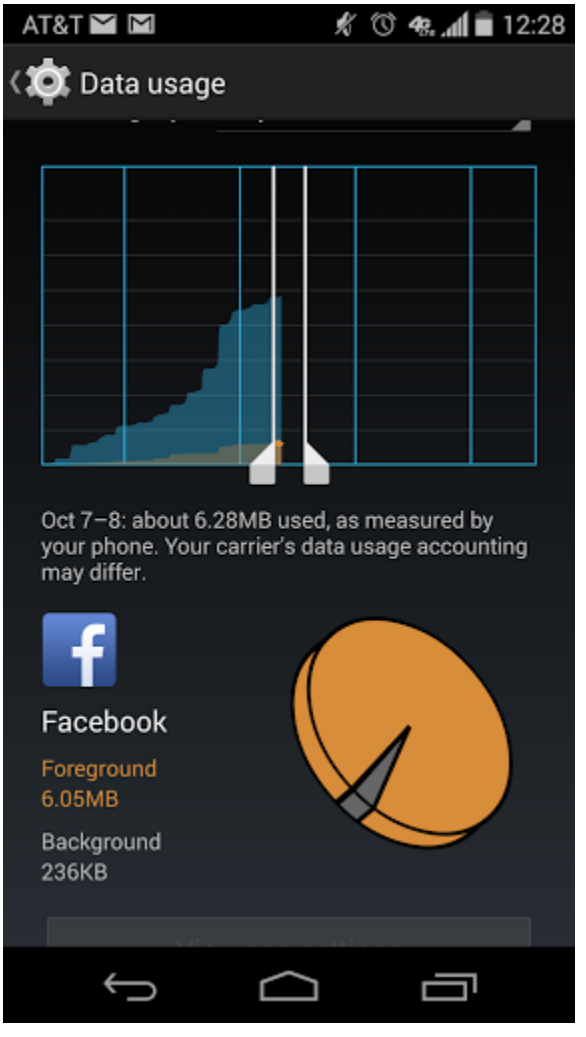


Figure 3-5. Facebook Data Details

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, 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.

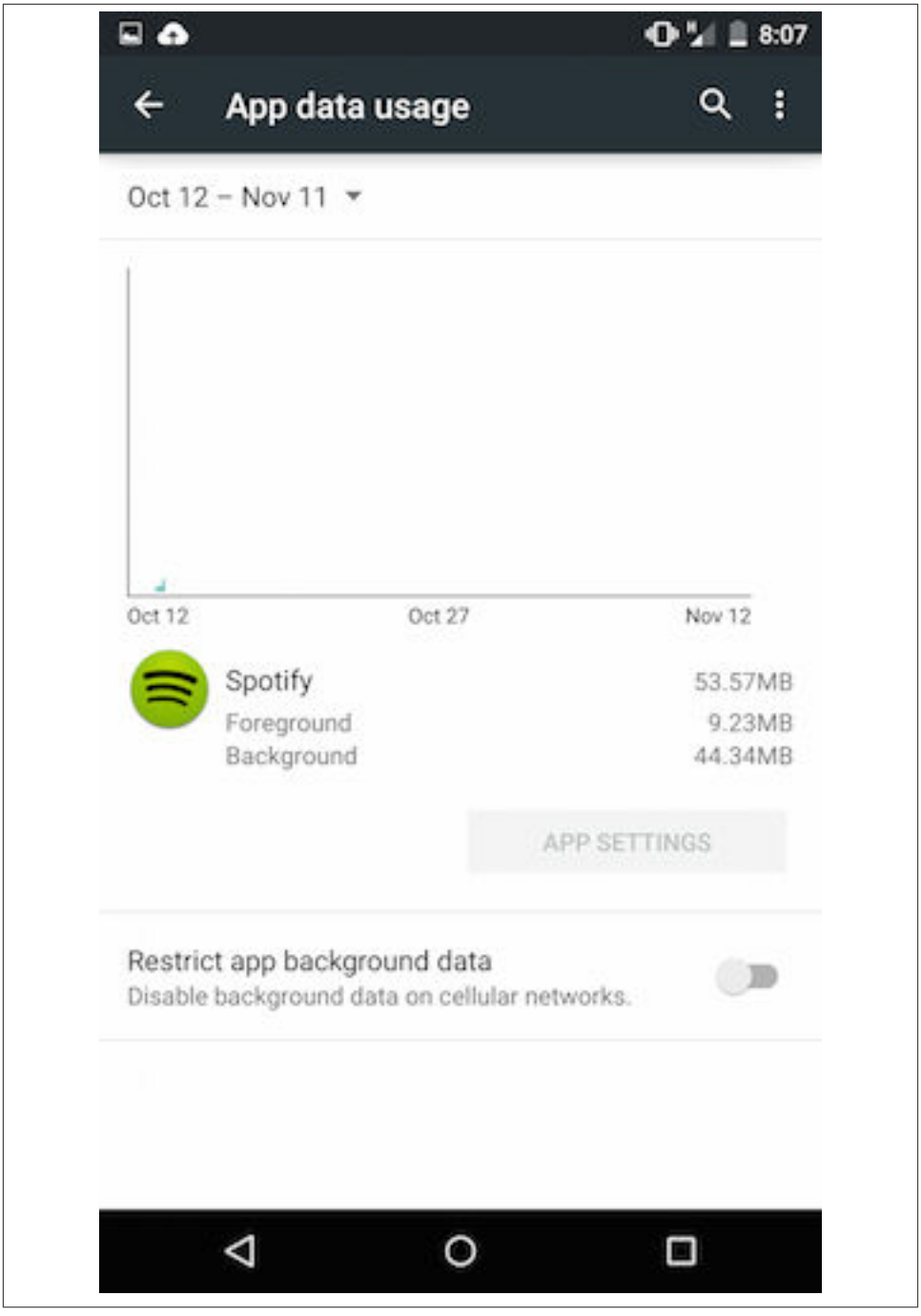


Figure 3-6. Spotify Lollipop Data Details

Prior to generating Figure 3-6, I had 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 40024 packets received (from the battery menu) delivered 53.57 MB of data, giving us useful values like 1403 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 applications 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 applications that are using more data than you 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 applications, but it is only a matter of time until these comparison tools become easier and more mainstream.

App Standby

In Android M, Google has announced a new feature called App Standby. App Standby will prevent infrequently used applications (apps that have not been used for several days) from connecting to the network or run 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 application 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 applications 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 thatWakelock reporting stopped working in all the Google Play wakelock monitoring apps). This has been expanded in Lollipop to provide more information and additionally some visualization tools have been added.

BatteryStats

Battery stats 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
```

BatteryStats

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

To get an idea of what a Battery Stats 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 few useful sections from the output stream. We'll be able to see some basic stats of the device - how long in 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 vs. 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 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 Wi-Fi available, but the Wi-Fi 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%) uptime
  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 section titled “Device battery use since last full charge” shows the estimates of battery usage % over the time period. This report is fairly boring, since 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 devices specific data, we begin to get more application specific data, starting with a list of all the processes 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 application name (we'll leave these processes anonymous here).

BatteryStats Per App Data.

```
  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 application, broken down by process. I am only displaying one application (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 realtime
    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 26ms.
- The vibration was 100ms 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 = 630ms. 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 analytics service was open for a few extra seconds after the chathead service 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 applications 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 application 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 a html document. At its simplest, you can run the following commands to create a webpage 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 application (and providing that your device is on Lollipop or newer). To begin, let's look at the report that is identical in both versions of Battery Historian (labeled Historian-Legacy in version 2.0.)

Let's continue evaluating the same trace, but now in the browser. The Battery Historian chart includes a LOT of data recorded by your device:

Figure 3-7. Android Lollipop Battery Historian (Overview)

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:

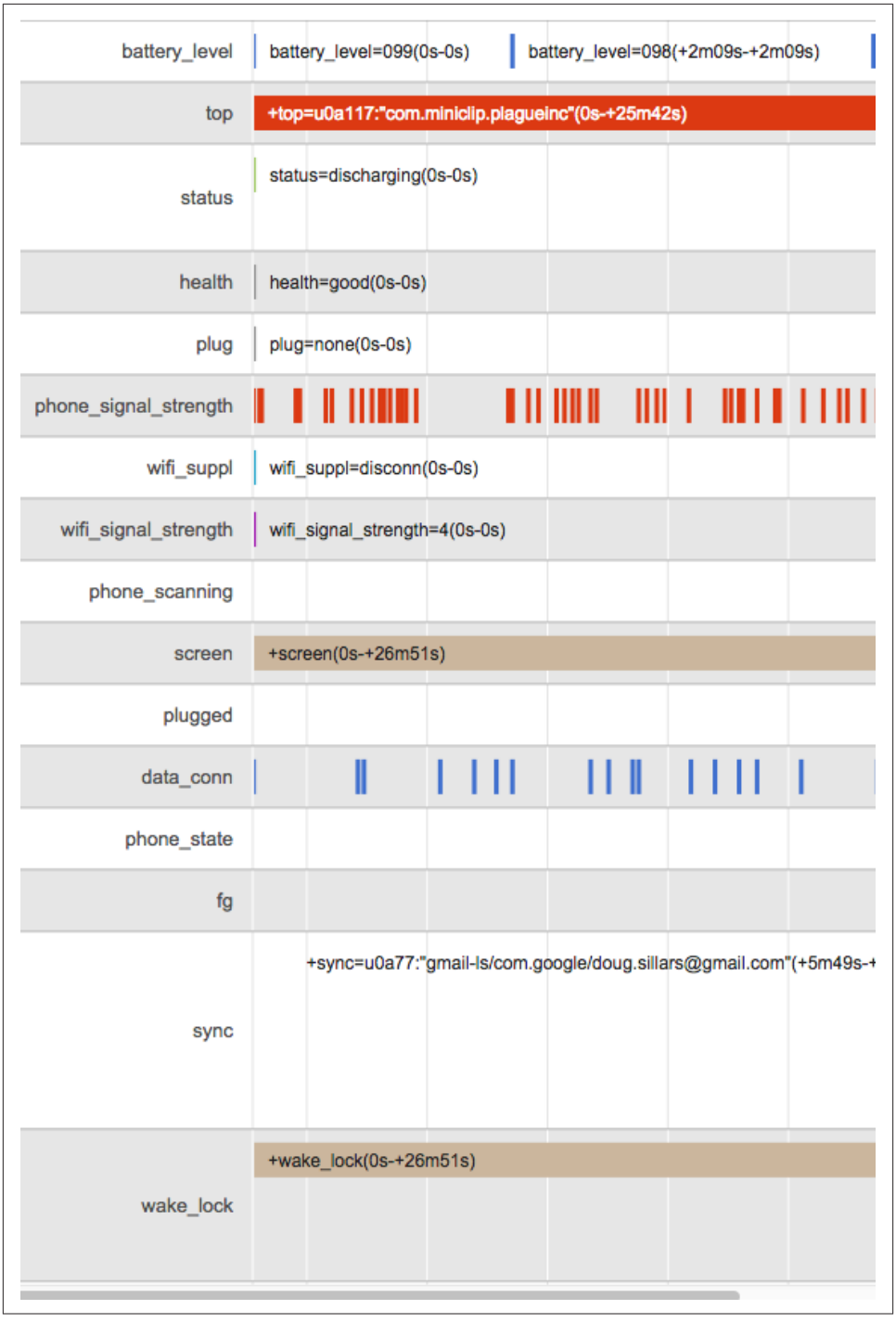
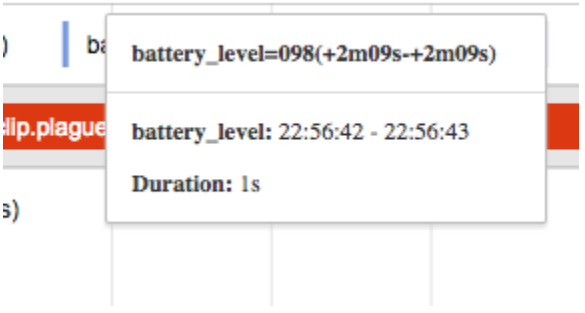


Figure 3-8. Android Lollipop Battery Historian Top View

In this chart, the white vertical bars indicate 1 minute intervals. For any item in the chart, mousing over provides more details.

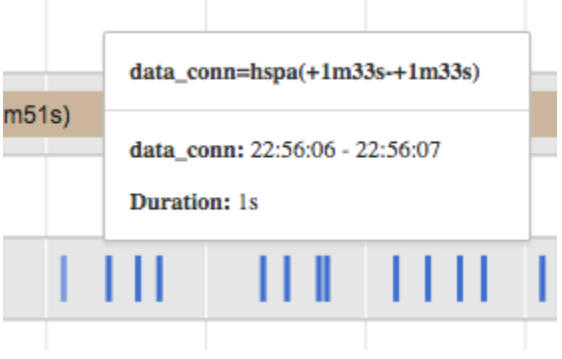
1. Battery level:



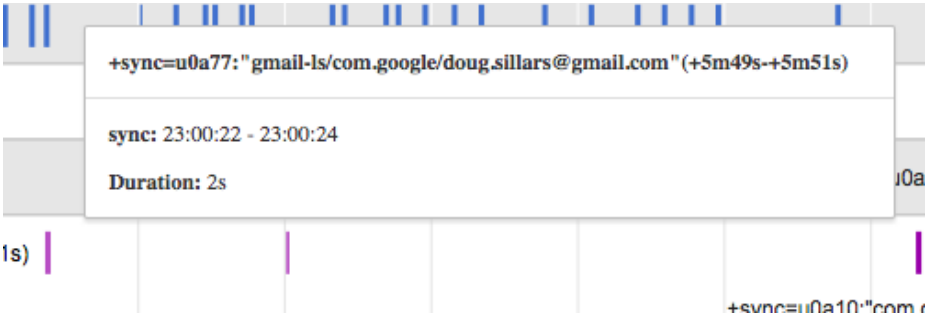
1. top: Lists the process that was actually on the screen. In this example - the game Plague Inc.
2. Battery info:
   a. Status: Battery is discharging (vs. plugged in)
   b. Health: Battery health from the Battery Manager API
   c. Plug: Is device plugged in?
3. Radio Information:
   a. signal strength: showing changes in cellular signal(in this case from poor, moderate and good)

   a. Wi-fi: Disconnected
   b. Wi-Fi: Signal (Wi-Fi signal is detected - even with Wi-Fi off, due to the Advanced Wi-Fi setting to always scan)

c. phone_scanning: If there is no signal, the phone will scan (using more battery power)
    a. screen: On/Off and duration on.
    b. Plugged: Power Source (similar to the Battery data above)
    c. data_conn mousing over the blue connections shows the cellular data switching from HSPA to HSPAP



1. phone_state: shows changes in cellular coverage, or if you get a phone call
2. fg: Foreground apps. Apps running in the Foreground are less likely to be killed to relieve memory pressures. We can see (just off the screen, that Facebook was coming into the foreground to process the message I received)
3. sync: processes syncing with the server



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 application is waking up the device often(and you will likely see your process name in the mouseover information), you should examine the fre-

quency 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 he 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, that 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 3-9. Android Lollipop Battery Historian Bottom View

1. wake_lock: There is one prolonged wakelock keeping the screen on during the game.
2. GPS: When the GPS radio turns on
3. Running: The phone was clearly running, as I was playing a game.

4. Wake_reason: This is when the device wakes from sleep. There are none on this screenshot, since 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 commone wake reasons are:
   a. qcom,smd-modem: Qualcomm Shared Memory Driver interacting with the modem memory
   b. qcom, smd-rpm: Qualcomm Shared Memory driver - Resource Power Manager
   c. qcom, mpm Qualcomm: MSM Sleep Power Manager: Shuts down clock and puts the device to sleep
   d. qcom, spmi Qualcomm System Power Management Interface: also working to put the device back to sleep.
5. Wake_lock_in: Here we can see what processes are running and causing the wake-lock or alarm to occur. In this 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 (5th row has a line of solid pink).

1. 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.
2. user: for cases where multiple users accounts might be used.

In this screen shot we are getting to what is waking up the mobile device. As mentioned above in "Wakelocks" on page 33, when your application wakes up the device, it leads

to battery drain. This screenshot is relatively boring, since 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 3-10), I force stopped an application that was using too many wakelocks.

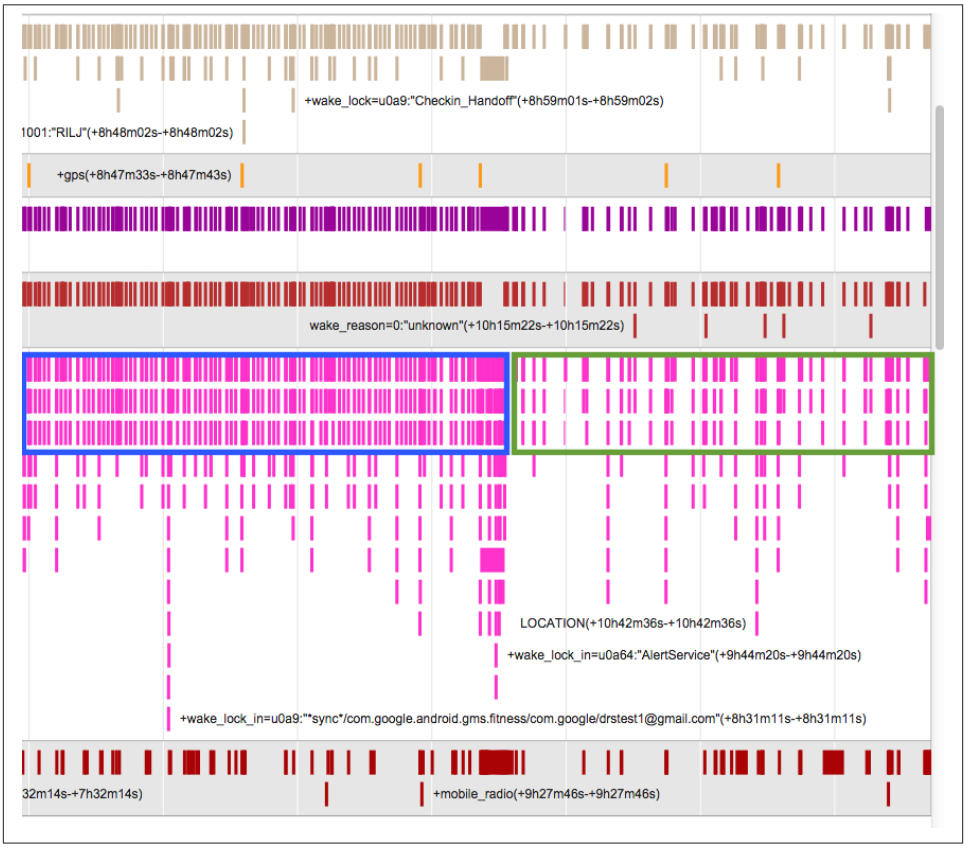


Figure 3-10. Finding Excess Wakelocks with Battery Historian

Qualitatively, the 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 repeat pattern to a number of wakelocks, as seen in the blue box. The application was firing at least 3 wakelocks a minute: the app, location, and accelerometer. These wakelocks are always 1 minute apart. In between the blue and green box, I stopped the ap-

plication, 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 hat you are not waking up the device to 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 above, 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 webpage 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:

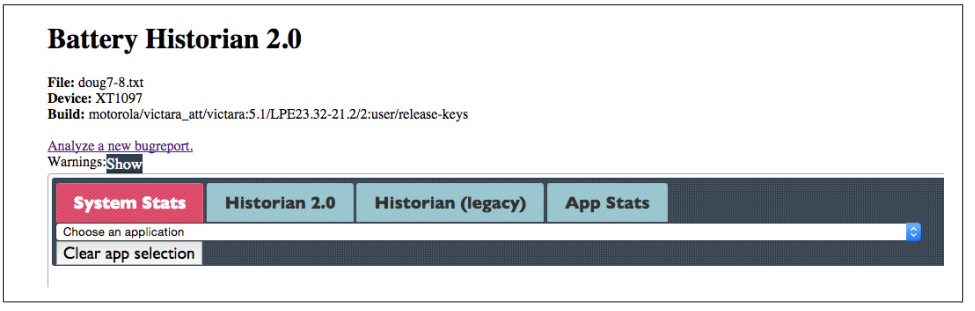


Figure 3-11. Battery Historian 2.0 Header

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: 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. Looking

closely, there are five underlined Metrics in this table. 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:

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 3-12. Battery Historian 2.0 Aggregate Stats

Looking at the process details for mobile radio time/app and kb/app, we can see what applications 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 of 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 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 application 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 application behaves, but also a very nice report format to share with your teams.

Stepping back to the entire trace for a moment, the Battery Historian 2.0 tab has a new UI showing how the wakelocks and device usage affect battery life.

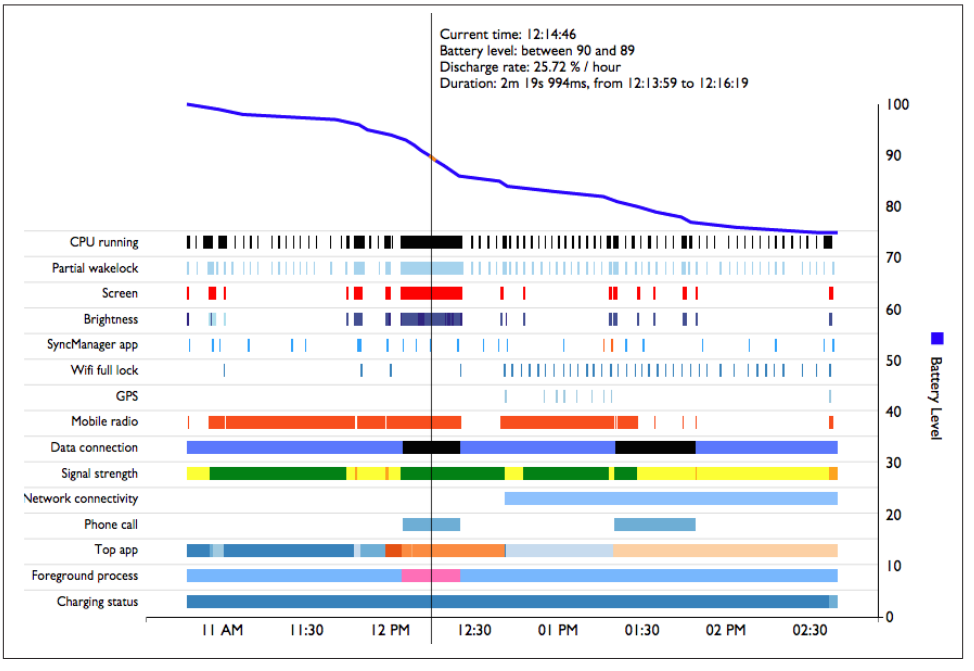


Figure 3-13. Battery Historian 2.0 Graph

This new chart has familiar bars for wakelocks, CPU, GPS, radio, etc., 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 closeup below, 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.

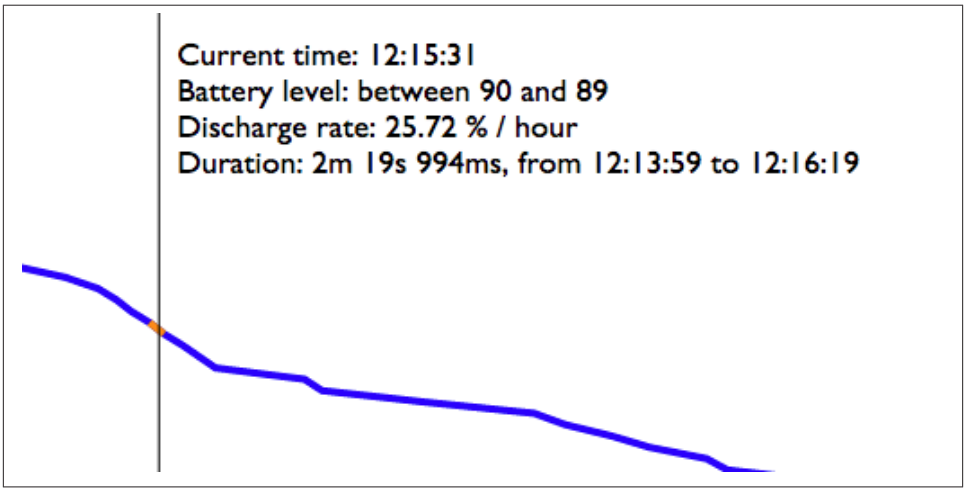


Figure 3-14. Battery Historian 2.0 Battery Drain Detail

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 application 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 application. Think of it as wakelock/alarms that "play well with others" API. While Wakelocks and alarms are application specific, the JobScheduler abstracts the device wakeups to the OS. Since alarms and wakelocks are sandboxed to your application, there is no way to coordinate these with the other applications installed on the device. If 5 apps wake up every 30 minutes, their alarms are unlikely to by synced, resulting in 10 wakeups per hour. Since jobScheduler abstracts the wakeup to the system, the system can piggybacks all of the scheduled jobs in an efficient way, so that there might only be 2-3 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 application 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 application, but uses less battery as a result. in the code snip below (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 application wakes up your customers' device every 15 minutes to check for updates on the server. That would be 4 wake ups an hour (96/day). What if your customers also have a weather app that updates every 6 minutes (10x/hour, 240x/

day)? The odds that your app, and the weather app connect at the same time is extremely low - since your timers are not synchronized, and there is no interaction between the applications. If both apps had used the JobScheduler API, the OS would coordinate to save power. In their Project Volta talk 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 this example, I have set the application to download an image every 60 seconds (the API will make a connection inside a timing of 60s, but not necessarily at exactly 60s.) This means that in ~ 14 minutes the app will ping 14 times.

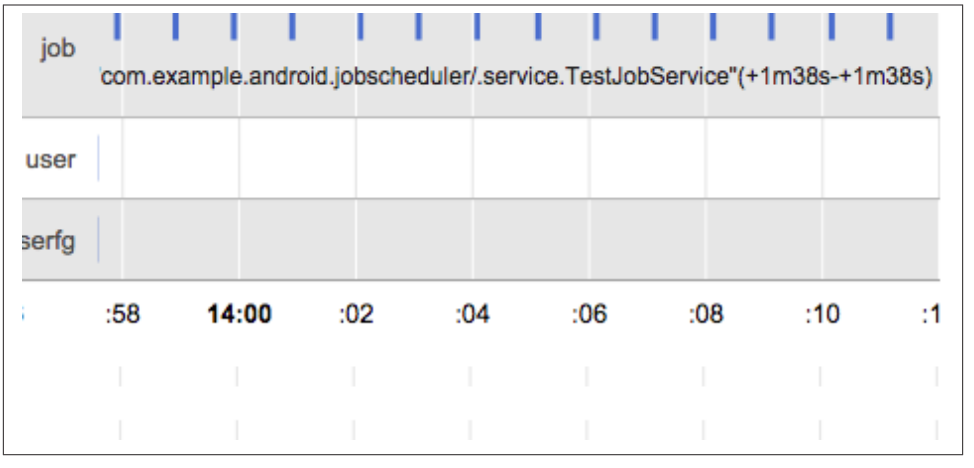


Figure 3-15. 60s Periodic Connection in Job Scheduler

The next graph shows a similar test, but where the periodicity is set to download every 150s (same scale), resulting in 6 connections over 14 minutes:

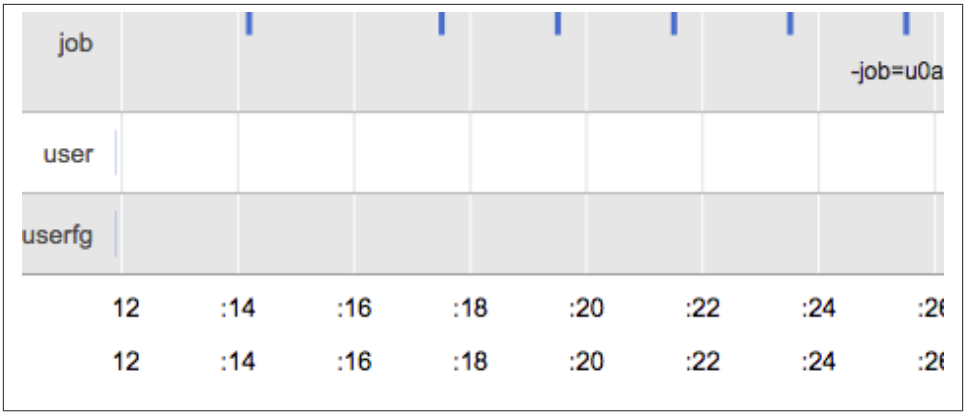


Figure 3-16. 150s Periodic Connection in Job Scheduler

With traditional wakelocks, we'd assume that if these were sandboxed applications 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 2 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 9 connections.

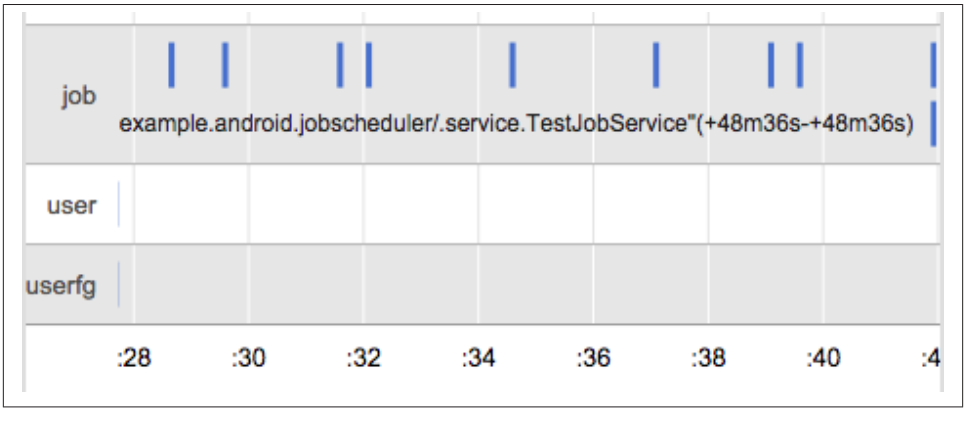


Figure 3-17. Synched 60s and 150s connections

Another cool feature of JobScheduler is the ability to repeat the job, but with a linear or exponential back off. 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: Linear (for slowly backing off) or exponential (faster backing off) of your job. The linear fallback takes the

current deadline, and adds fallbacktime*(number of failures -1). In the example below, the deadline was 20s, and the fallback delay was another 20, so each subsequent ping adds 20s (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 application data is being updated for your customer, but in a way that uses the network less - saving battery.

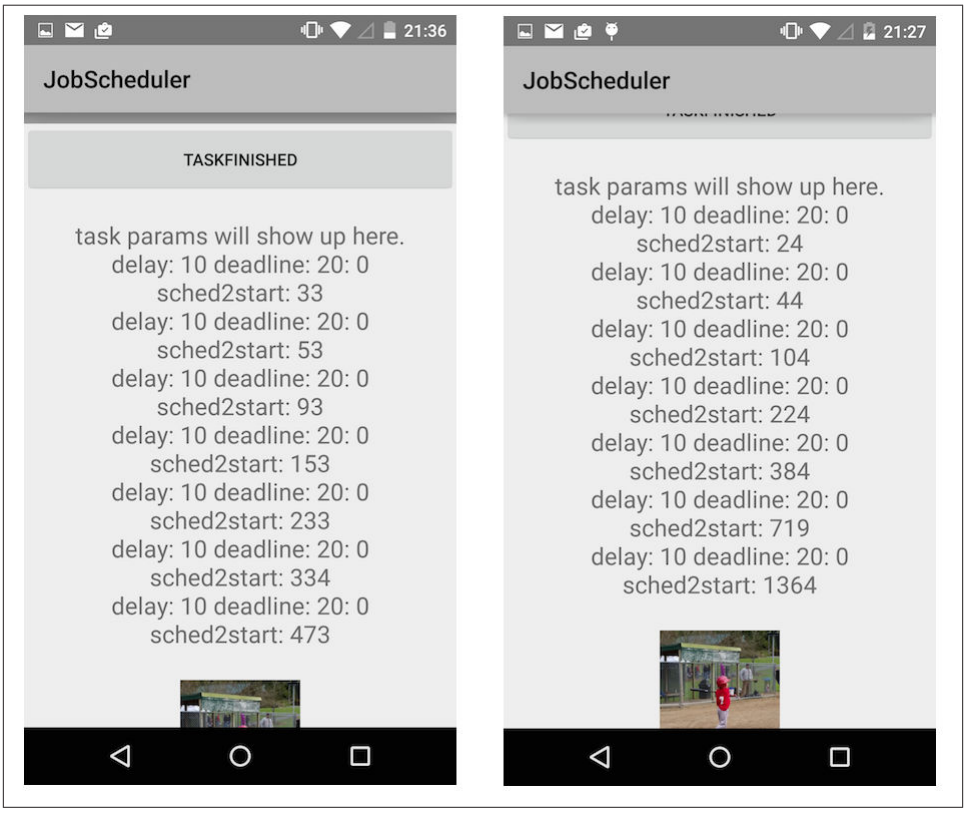


Figure 3-18. Screenshot of the jobScheduler app with linear (left) and exponential (right) fallback

It is clear that allowing the OS to schedule non-critical jobs is a great way to conserve battery life in your application. 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 application 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), how customers can discover apps that are causing issues (and help you make sure your app doesn't appear to 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 application 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 application is likely influenced by designers, developers, usability studies, testing, and 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 our application is, was your UI design built to be performant?

As a developer, your task is to work with the UI/UX 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 delays of:

- 0-100ms feels instantaneous to the user
- 100-300ms sluggish
- 300-1,000ms machine is working
- 1,000ms+ your customer feels a context switch. Delays are noticeable.

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 "1,000ms to Glass." If your webpage 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 3-4s. Applying the same argument to apps tells us that the faster you can get your application to start, the better. In this chapter, we'll focus just on the UI loading. While there may be tasks that must run in the background, files to be downloaded from the internet, etc. 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). Since the screen is refreshed every 16 ms (1s/60fps = 16ms per frame), it is crucial to ensure that all of your rendering can occur less than 16ms. 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 16ms. 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. I'll walk through a few of the improvements that have been added in various releases of Android to improve the user interaction.

- In Gingerbread and earlier devices, 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 applications "buttery smooth." By improving timing 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 walk through all of these changes, the tools introduced, and what they all mean to the average Android Developer. As you might imagine, the goals from these updates were to:

- Lower 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 applications for rendering performance issues. As we walk through the different ways to improve application performance, we'll use these tools with example applications to explain how they work.

With that, let's get started!

Building Views

I am assuming that we are all 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 the figure below, you can see a simple application with 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.

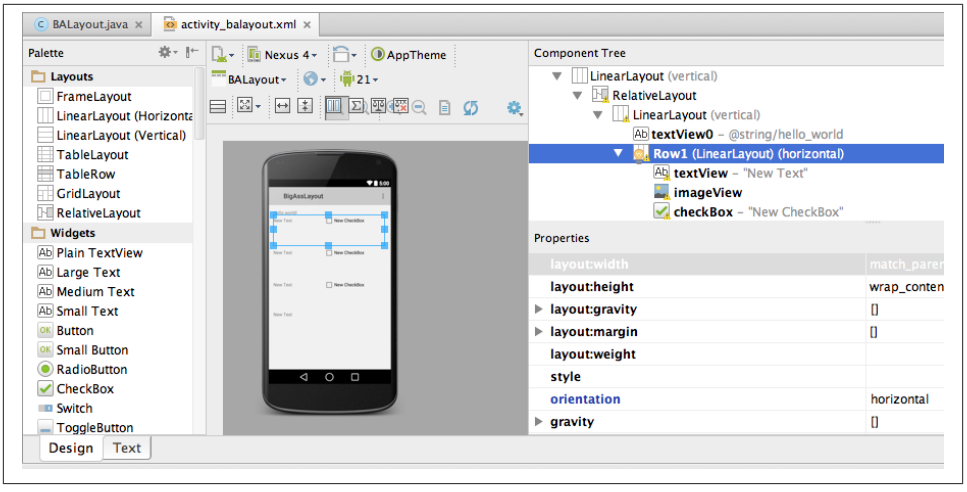


Figure 4-1. Design View of an Application Layout

For each view in your app, Android goes through 3 steps to render on the screen: measure, layout and draw. If you imagine your XML layout hierarchy in your application, the measure starts at the top node and walks the rendering 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 3 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 re-measure 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 re-measurement 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 (4 remeasures with 2 nested views, 8 with three, etc.). We'll see a dramatic example of remeasurement in Figure 4-8

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, the 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 86. The more views your app has, the more time it will take to measure, layout and to 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 16ms 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 View tool in Android Studio Monitor can greatly help you visualize the views in your Android app to resolve these issues. (if you have not yet discovered Monitor, it is a standalone application 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 view is a great tool to investigate the construction of your view XML. It is available in Android Studio Monitor tool, and requires a device with a developer build of Android on it. See "Rooted Devices/Engineering/Developer Builds" on page 14 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 this device, you app will likely render well on all Android devices.

When you open Hierarchy View, there are a number of windows: at the left, there is a tab "Windows" 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 apps' 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).

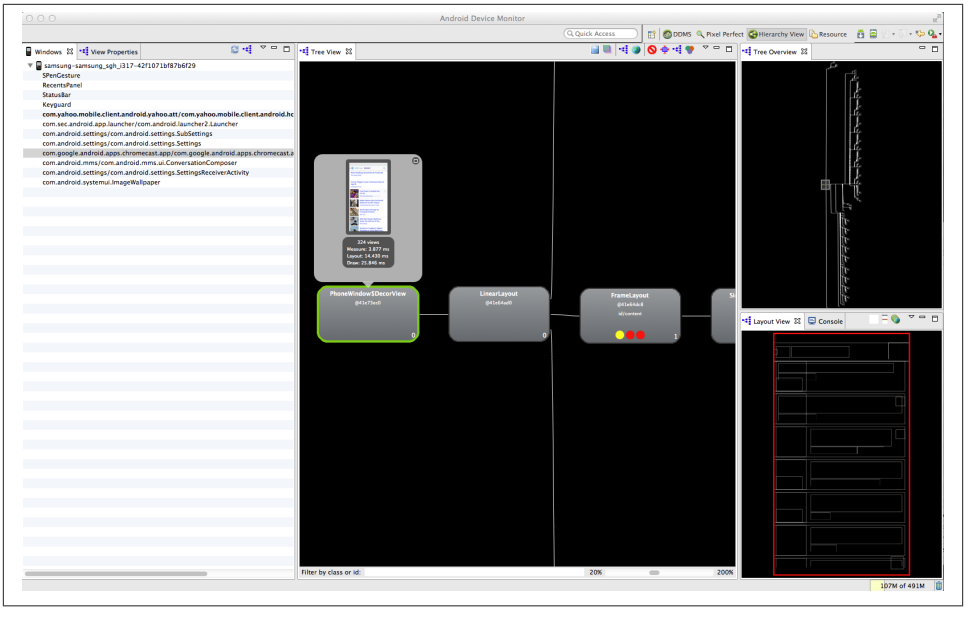


Figure 4-2. Overview of the Hierarchy View tool, using the view tree of a news application.

Inside the central closeup 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 popup 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 this case, I chose the top view to obtain the timing to create the entire view).

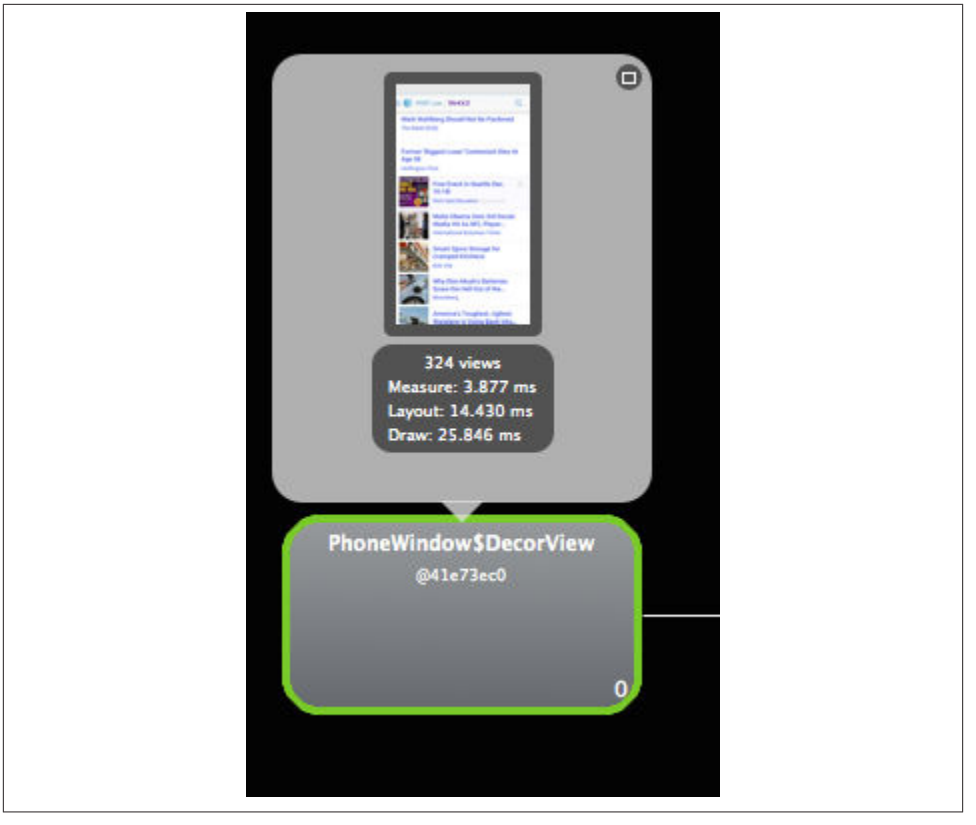


Figure 4-3. Render Times of a View

The topmost view for the article list uses 324 views, measures in 4 ms, layout in 14ms and draws in 26ms (~44ms total). In order to reduce the time to render these views, it makes sense to look at the Tree Overview of the application, 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 us good, since the "depth" of your view XML can have detrimental effects to rendering time. Even with the flat XML, there is still a 26ms draw time, there may be times where this view is janky, and that optimizations should be considered.

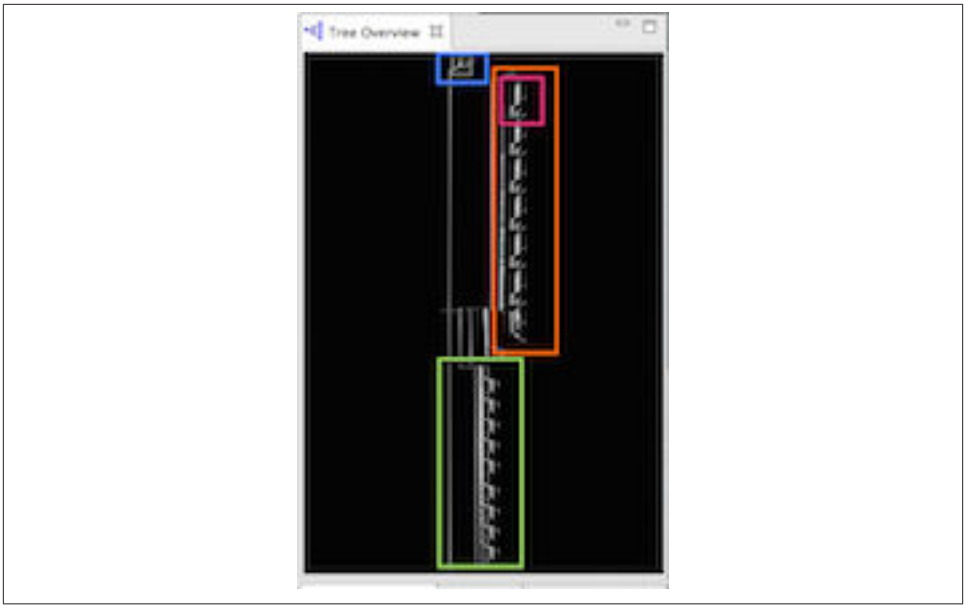


Figure 4-4. Tree Overview

Examining the Tree Overview of a news application's list of articles, there are 3 major regions: the header (in the blue box at the top of the views), story lists (the top long orange box, with a single story highlighted in pink - there are 7 repeats of this internal headline structure), and the side pull out navigation bar (bottom green box). The header takes 13 views, the story list 190 (each headline uses 21 views) and the navigation drawer ~120. 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.

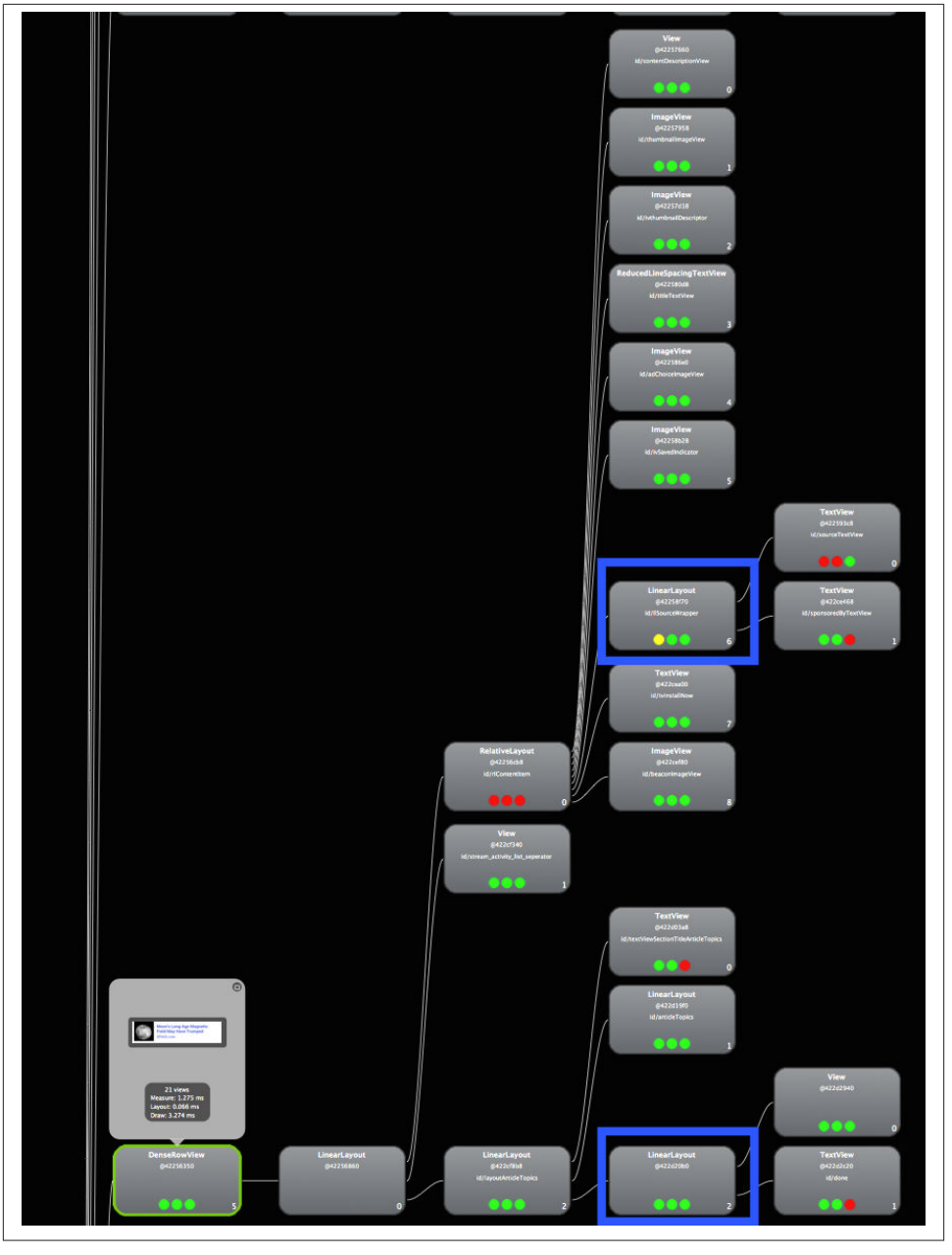


Figure 4-5. Examining a View Tree

Looking closer at a headline, we can look at the 21 views that make up one headline in the list. Each headline has 5 levels of hierarchy (seen as vertical columns), and it takes

1.275 ms to measure, 0.066s to Layout and 3.274 to Draw. The 5th layer of hierarchy is fed by 2 LinearLayouts in the 4th level (highlighted in blue). If the layout described in these 2 LinearLayouts could be described in their parents (in the 3rd level), on entire layer of rendering could be removed.

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 RelativeLayout in the 3rd level of the tree (about 1/3 of the way up the tree). This is where the actual headline list is built. (In fact, the other views that feed off of the 3rd level (in s straight line form the parent views) feed nothing to the screen (in the "View Properties", there is a Layout option, and it shows a height and width of 0,0). Discovering and removing these views could help to speed up the rendering of this application. After seeing their Layout in Hierarchy View, the developers worked to optimize their view:

Figure 4-6. Updated View Tree

This redesign has a bigger image and has added share buttons. Yet is flatter, has fewer views, and is nearly 400ms faster to display!

The "Is it a Goat?" sample app on Github 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 application. We'll walk through several steps of optimization in this application, 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 application reveals a few things:

Figure 4-7. Hierarchy View of the Unoptimized Goat app

There are 59 views in this simple application. However, unlike the news application 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. 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 7 views in a row that only add depth to the application, 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:

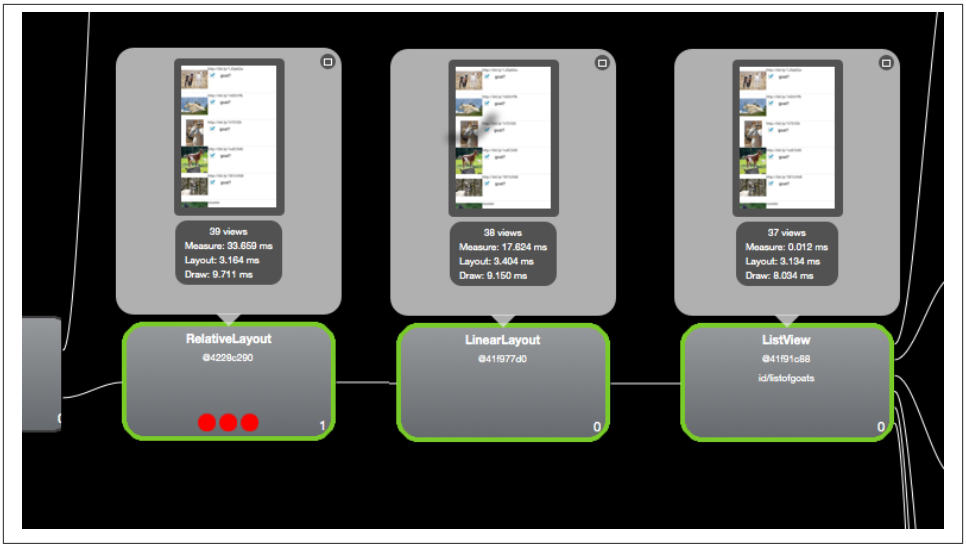


Figure 4-8. Remeasurement in Hierarchy View

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 6 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 re-measurement loop. The measure time jumps 3 orders of magnitude to 18.624ms. The Relative Layout to the left of the LinearLayout redoubles the measurement time to 33.659ms. By the time the measurement cascades through the additional parent views (those additional views in the red box in Figure 4-7), the total measurement time is over 68ms. 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.) Applying the "Optimized layout" view to the application removes four layers if view depth from the red box Figure 4-9, which speeds the but the real savings comes from removing just one view.

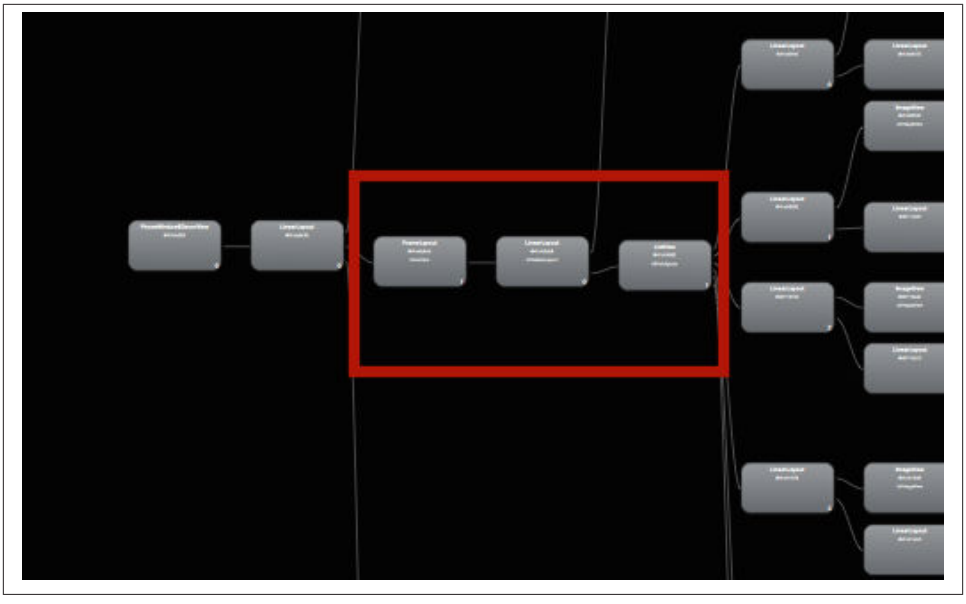


Figure 4-9. Removing Hierarchy Depth

A further optimization to remove view depth can be done by looking at the rows of goat data. Each line of goat information has 6 views, and there are 6 rows of data visible on the screen (one such row is highlighted in a purple box at the bottom right of Figure 4-7.) Zooming in Figure 4-10, we see two sequential linear views that only add depth to the row of goat information ("Slow XML" view). The initial LinearLayout feeds directly into a RelativeLayout, but adds nothing to the display:

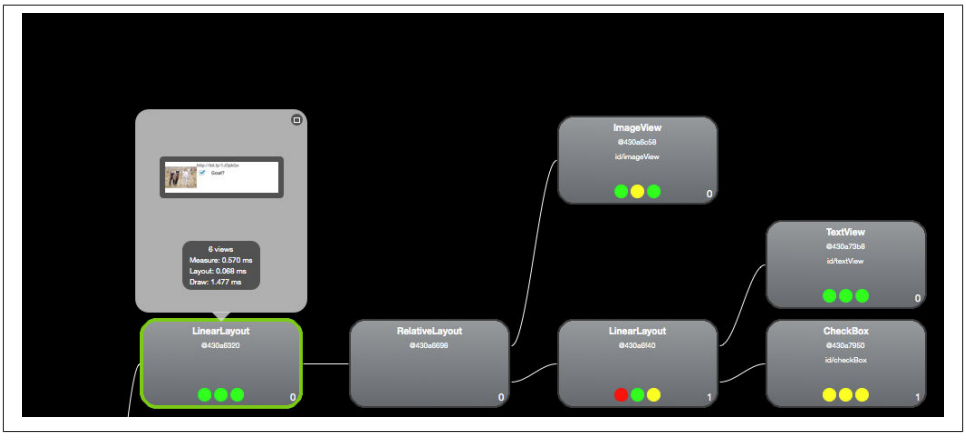


Figure 4-10. Unoptimized Hierarchy View of Goat Row.

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.) When I did this, the depth was reduced from 4 to 3, and the display rendering is faster:

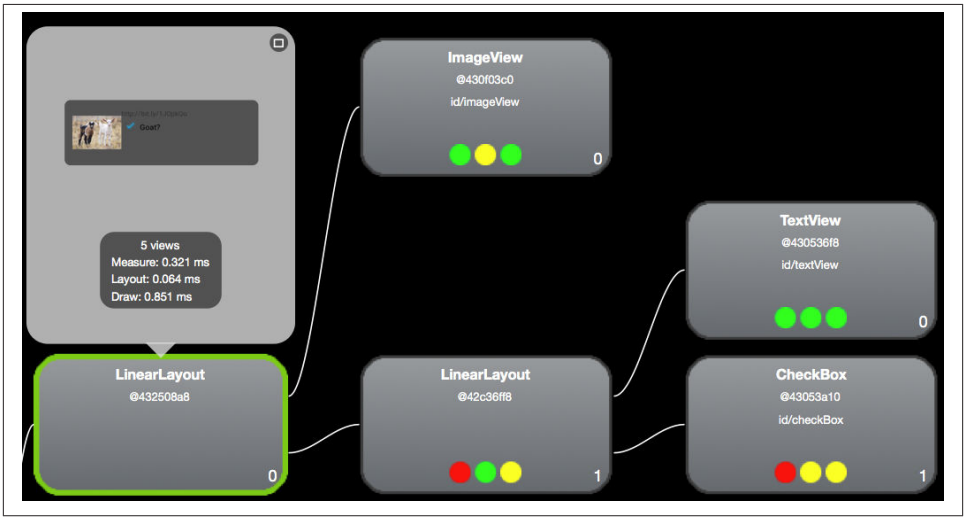


Figure 4-11. Views Optimized by Removing RelativeLayout.

However, this is not the fastest optimization. By removing the LinearLayout and reorganizing the RelativeLayout to handle the entire row of information, the view depth is reduced to two. 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.

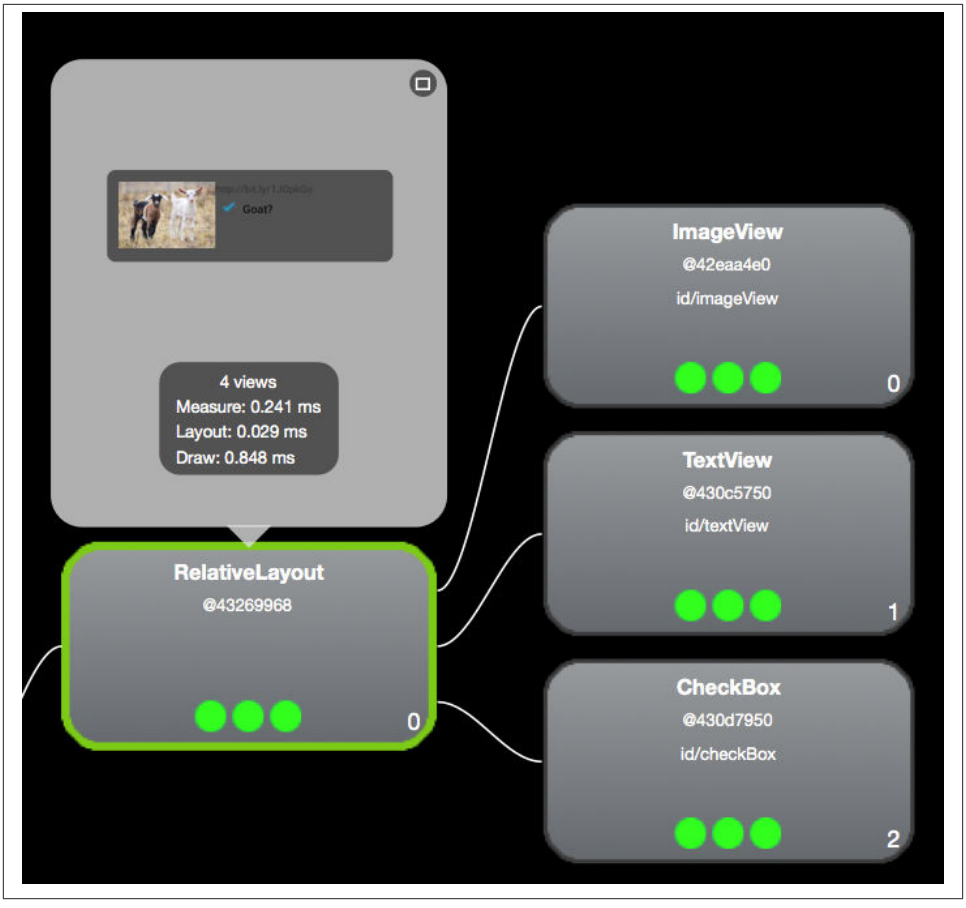


Figure 4-12. Views Optimized by Consolidating into RelativeLayout.

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 ~1ms of rendering from each row of information, we can pull about 6 ms from the entire render time (assuming 6 rows of data on the screen). If your application has jank, or your tests show that you are close to the 16ms 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 sub-layout 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> </merge> tags. This will remove the extra layer of hierarchy from your application.

As an exercise, download the "Am I a Goat?" application 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 application, 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:

1. Save the any view from the tree as a png (icon is a stylized diskette)
2. Photoshop Export (described in "Overdrawing the Screen" on page 86)
3. Reload the view (2nd purple tree icon)
4. Open large view in another window(globe) - this has an option to change the background color to better view if there is overdraw
5. Invalidate the view (red line with bar through it)
6. Request view to layout
7. Request view to output the draw commands to the LogCat (yes, the 3rd use of the purple tree icon)
    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 use analysis tool to optimize the View tree of your application - potentially shaving tens of milliseconds from the render time of your Android application.

Asset Reduction

Once your application 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 use 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 vs. unstarred, online vs. 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.

In 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, so 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 application has multiple layers or view 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 a very useful way to quickly look at applications 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).

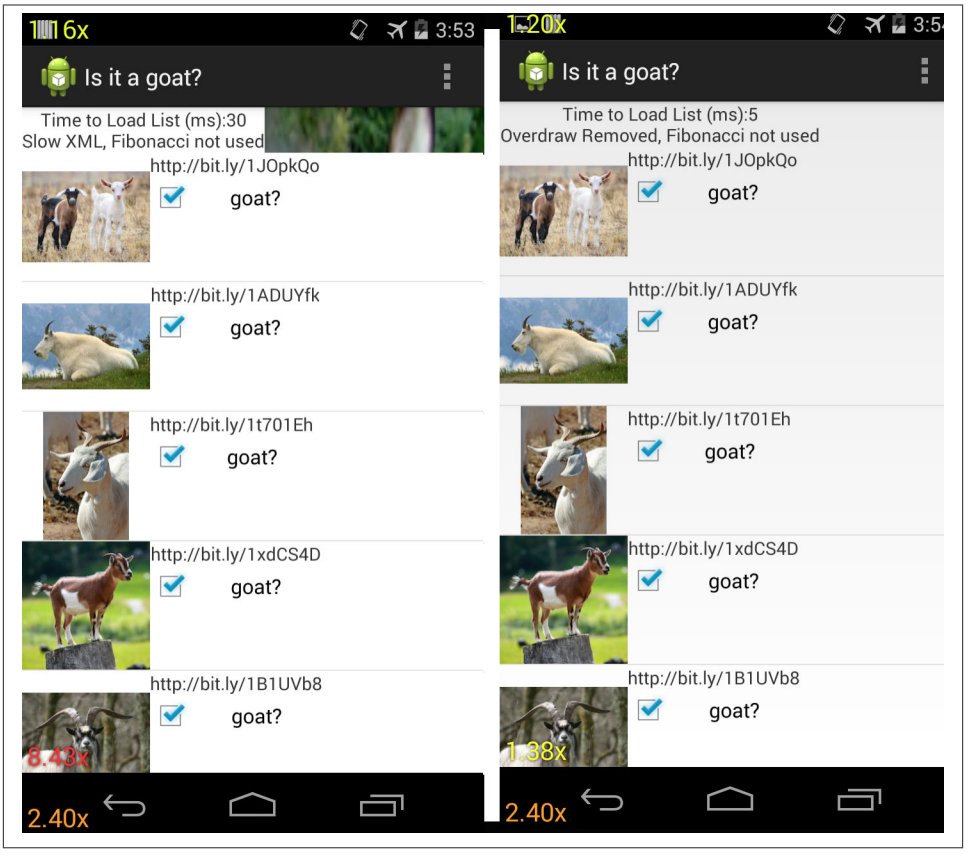


Figure 4-13. Overdraw counter for Unoptimized (L) and Optimized ® views of the same app

The screenshots above are again from "Am I a goat?" The overdraw counters can be seen on at the lower left. There are 3 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 application on the left has 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 applications without impacting the user experience too much, it does not help you understand where the overdraw issues lie.

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 application, showing you the total amount of overdraw in each region of the app. (For those developers who are colorblind, the KitKat release offers a setting that allows you to similarly debug that is colorblind friendly.) 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 4-14 you can see the overdraw areas rendering of the "Is it a Goat?" app before and after optimization. The menu bar of the application 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 3 layers of drawing were removed! There is now no overdraw around the text and in the *blank space.*

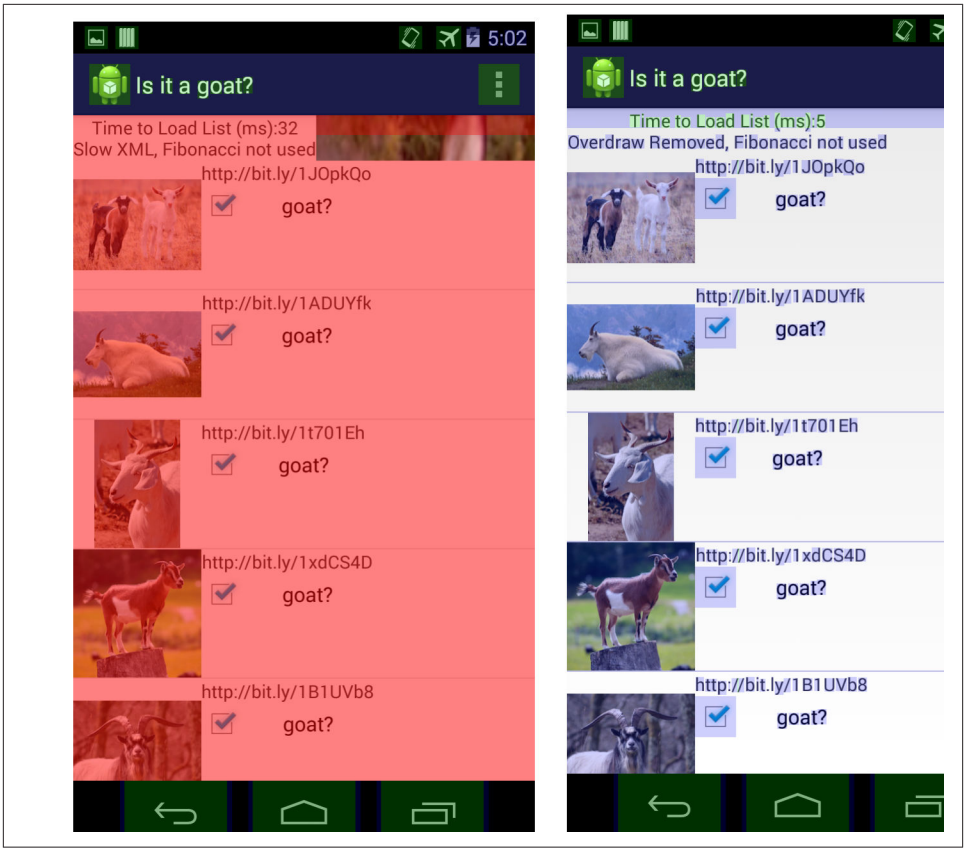


Figure 4-14. Overdraw Colors before (L) and after ® Optimization

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" vs. "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 application is to save the view hierarchy as a Photoshop document (the 2nd option on the Tree View) in Hierarchy Viewer. You do not need 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 applications, 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 application have a small eye logo next to the layer. In Figure 4-15, you can see that I have begun to peel back the views at the top of the goat application (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.)

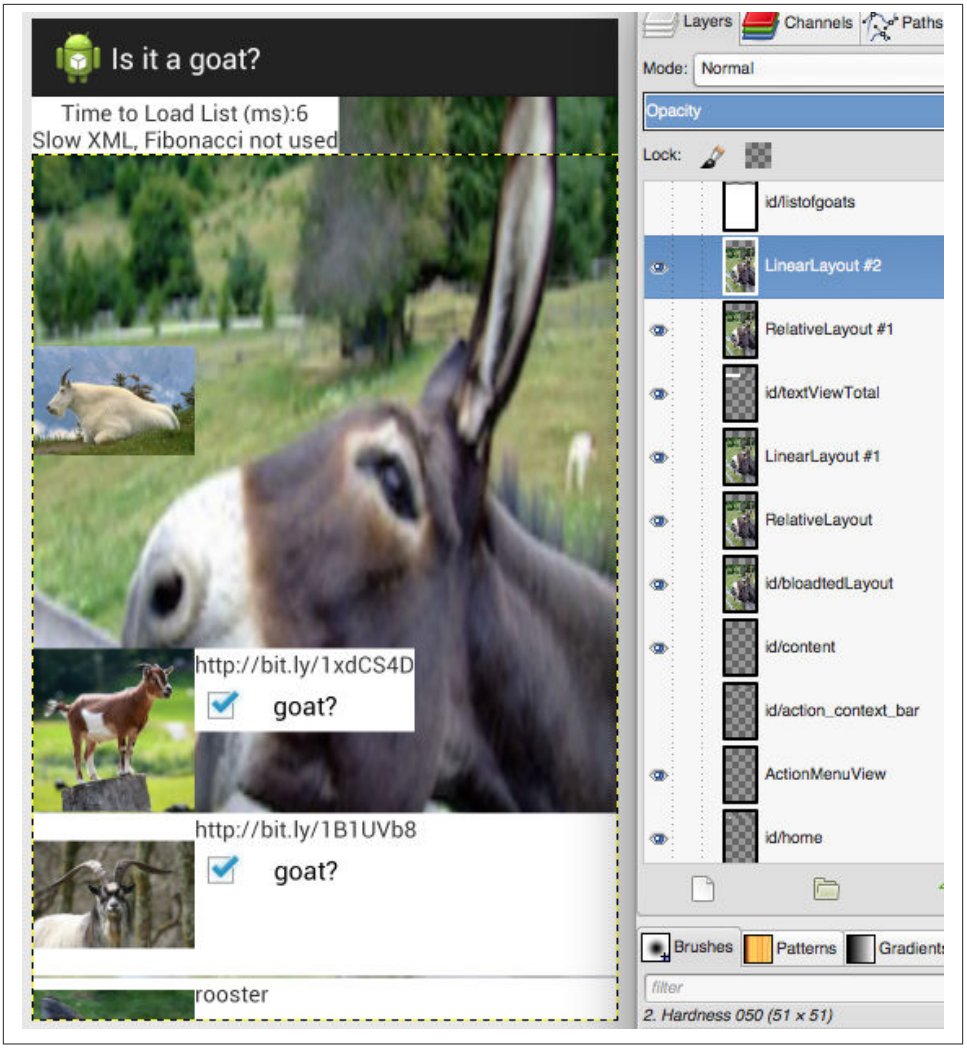


Figure 4-15. Visually Peeling Back Views

Another way to visualize the "peeling back of the views is shown below In the figure below, 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 2 rows of goat pictures and layout, revealing that under each row of goat data there is a stretched picture of a donkey. Below the 6 or 7 stretched small donkey images, there is a white backdrop (seen in the rightmost image on the top row with 2 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.)

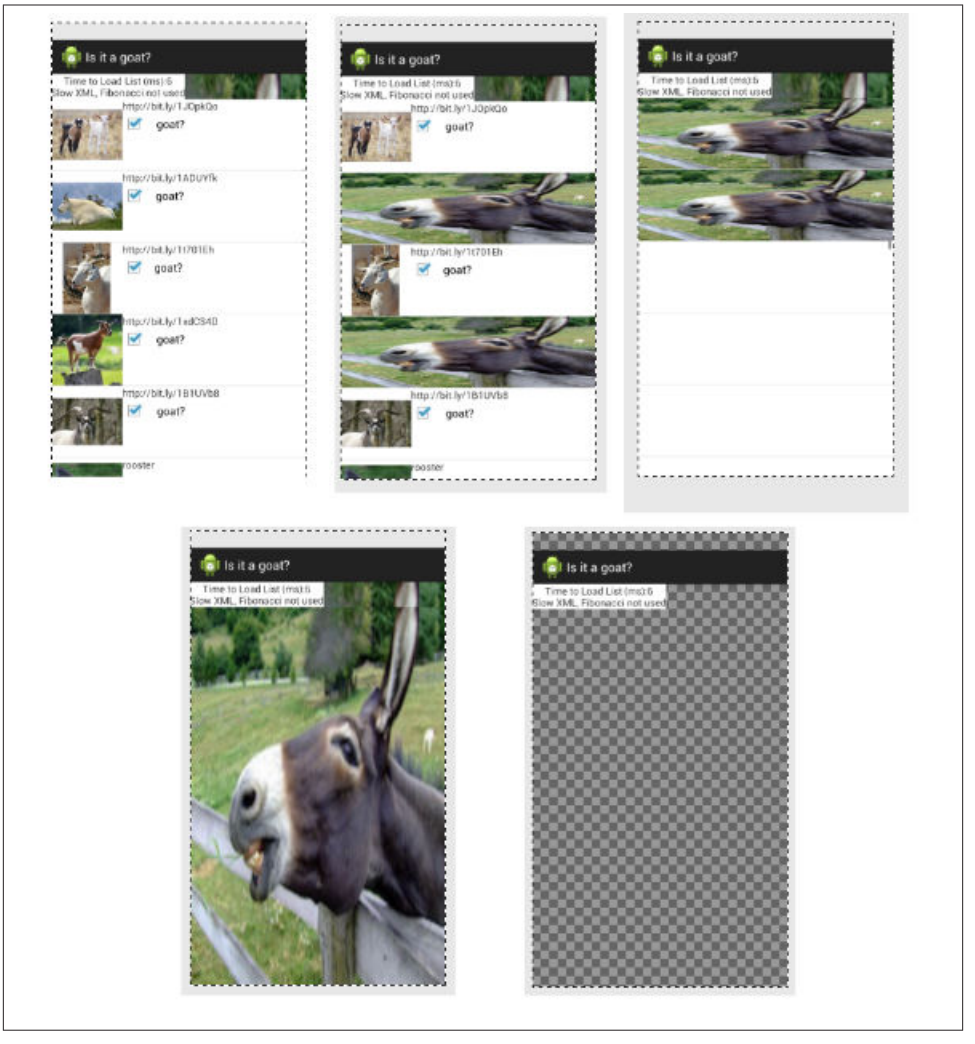


Figure 4-16. Looking at the Layers Visually

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 goat application will not be felt by KitKat and newer users). This will improve the draw rate for applications 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 above, 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 application, 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 log file (adb shell dumpsys gfxinfo), or you can display the GPU rendering as a screen overlay in realtime on the device (available on Android 4.2+).

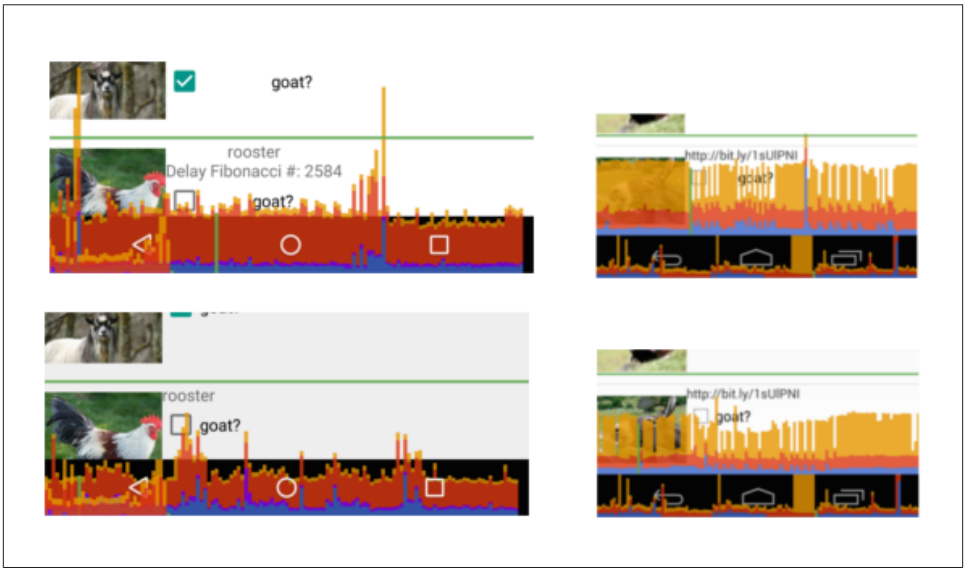


Figure 4-17. GPU Profiling Lollipop (left) and KitKat (right) of the unoptimized goat app (top), optimized (bottom)

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 4-17, 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?" application. 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 16ms 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 16ms 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), execute (yellow). In KitKat and earlier, the purple prepare data is not broken out, and is included in the other metrics (hence only 3 colors appear in the KitKat GPU profile screenshots.)

Comparing the GPU data from the Nexus 6 to the Moto G brings us back to the topic of device testing. The unoptimized Goat app (top row) in Figure 4-17 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 example below, 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 ~4ms longer than the Nexus 6. This goes to show that testing GPU rendering is best done on your lower powered devices, since they are more likely to have issues rendering your views without jank.

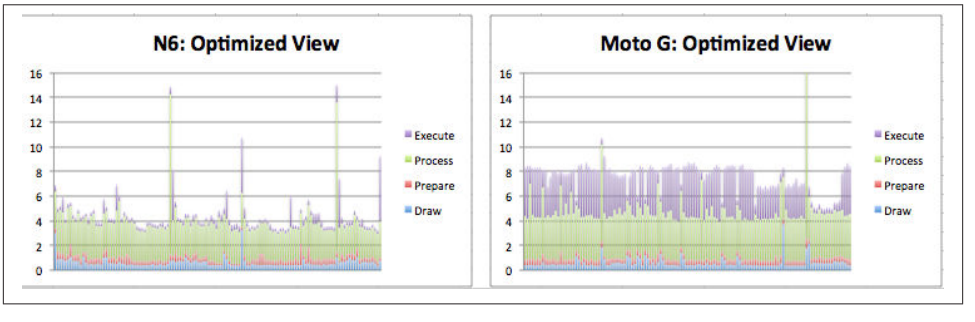


Figure 4-18. GPU Profiling Lollipop (left) and KitKat (right) of the Optimized views

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 application (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 application.

GPU Rendering in Android M

In Android M, 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 application:

```
** 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 application 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 16ms. 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 M to the gfxinfo library of test tools is adb shell dumpsys gfxinfo [package] 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 < 2ms))
- animation evaluation (should be < 2ms)
- layout+measure
- View draw
- sync phase (if > 0.4ms, indicates many new bitmaps being sent to GPU)
- GPU Work time (overdraw draw time will appear here)
- total Frame draw time

There are 2 tabs in the worksheet with sample data - both form the "is it a goat?" app: goat-optim and goat-slowXML. Looking at the data from the goat-slowXML sheet, we can see a few frames (in purple) where the total frame draw exceeded 16.6ms. 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 applications 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.

vsync - intended vsync	input event time (ms)	evalute animators	layout + measure	view.draw() time	sync phase time	gpu work time	total frame time
0	0.990573	0.088542	0.08625	0.909531	0.602396	6.557604	9.655536
0	12.068594	0.026771	0.047292	2.656406	0.772708	3.606667	19.487705
0	0.362864	0.131511	0.041093	2.004532	0.287552	10.933281	14.067426
0	14.328802	0.040677	0.04599	2.525573	0.677031	4.103385	22.192303
0	0.333333	0.029219	0.030885	3.613386	0.314895	11.57	17.446451
0	0.034948	0.351823	0.087135	0.93724	0.542656	10.610677	12.982787
0	0.037657	12.815052	0.04651	3.395417	0.624896	3.434323	20.810217
0	0.02	0.184427	0.033333	3.112344	0.250573	9.730469	14.14624
0	0.032291	12.791303	0.04776	3.381979	0.642344	4.273437	21.56195

Figure 4-19. Data From gfxinfo framestats

Beyond Jank (Skipped Frames)

There are times that the GPU profiler does not show a jank event crossing the 16ms 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 log files in the DDMS view. It is easier follow logs form 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 log files for a warning like this:

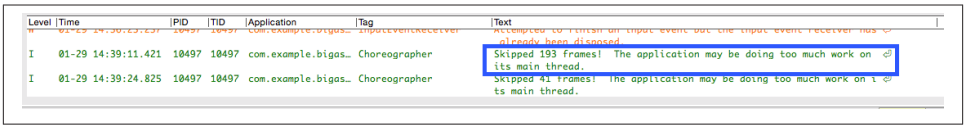


Figure 4-20. Log error showing Skipped Frames

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 application, 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 application 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 application 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 form Android Studio or the command line.) The Systrace icon is a stylized green and pink graph icon (marked by a red oval in Figure 4-21), and when you press it, it opens a window with a number of options:

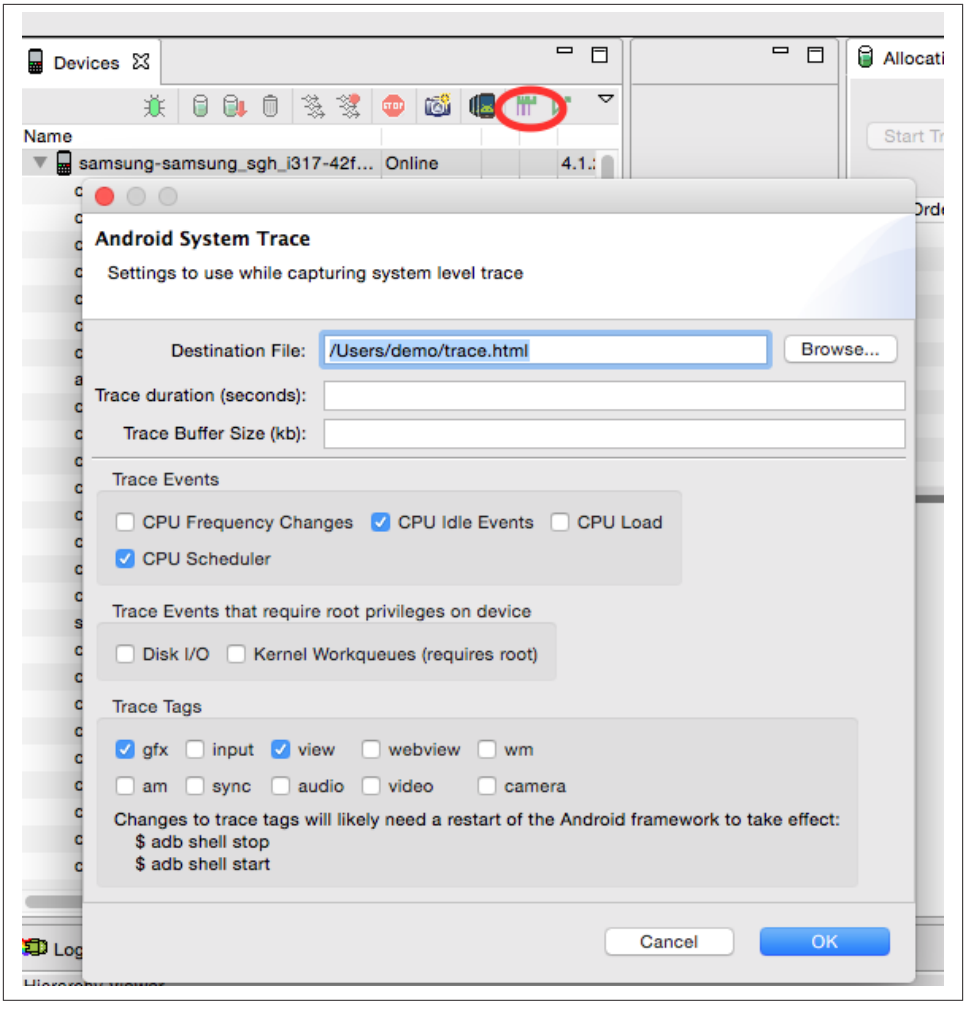


Figure 4-21. Starting With Systrace

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 4-21). 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.) Since the trace is extremely detailed (and measures all features to the sub millisecond timeframe), it is really used to diagnose one issue at a time, not a tool to get a holistic view of your applications performance.

Much like the "Battery Historian" on page 49 in Chapter 3, the output from these traces is overwhelming (and we only picked 4 of the options available!). Scrolling can be per-

formed 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 collapsable 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 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.

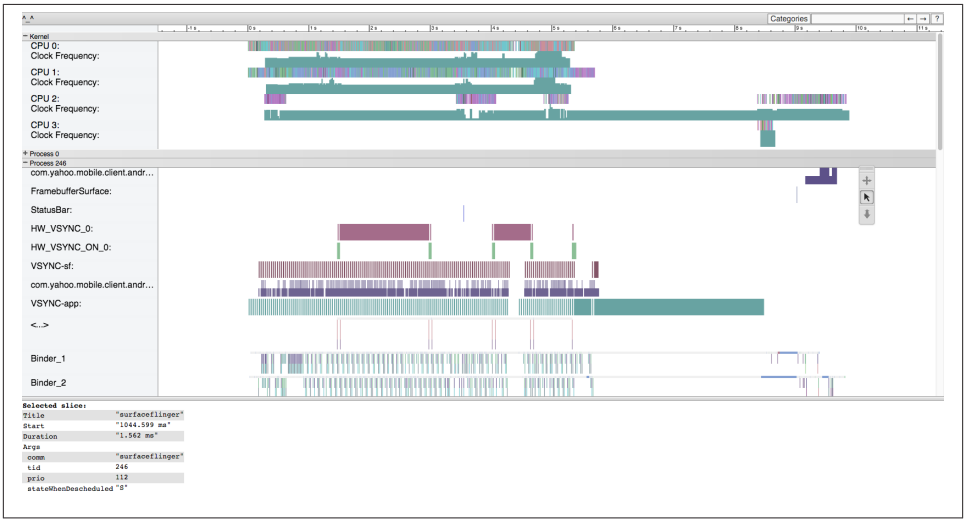


Figure 4-22. Starting With Systrace (Lollipop)

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 109.

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 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 4-23 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 16ms (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 16ms 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).

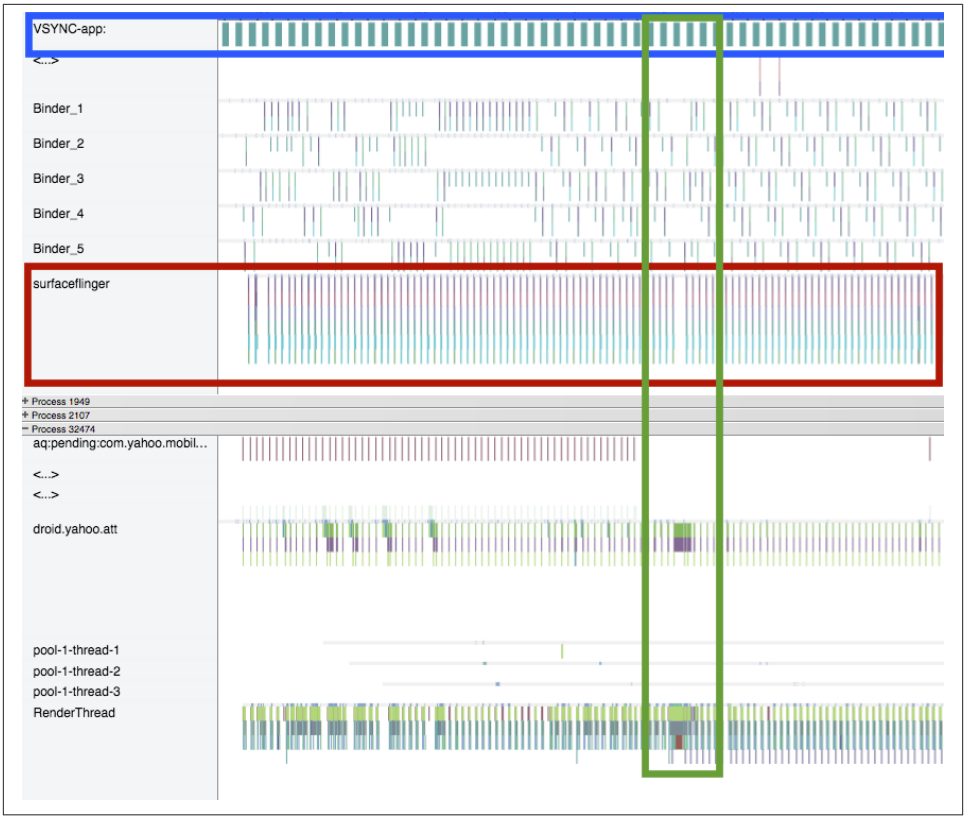


Figure 4-23. Digging into Systrace Jank (Lollipop)

The bottom section of Figure 4-23 shows details about the application. The 2nd 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 1/3 of the way through the trace), indicating that something in the app may have been the cause of the jank. Each application 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 4-24, 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.

1. Red box: droid.yahoo.com is finishing up a measure of the views for the screen. These are passed to:
2. Orange box: RenderThread. Here the Application:
   a. Draws the frame (light green)
   b. Flush the Drawing buffer. (gray)
   c. Dequeue the buffer (in purple)
   d. Sends this to a buffered list of views.
3. Yellow Box: com.yahoo.mobile.client.andr…
   a. 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.
4. Green Box: VSYNC-sf alerts the surface flinger that it has 16 ms to render a screen. The brown bar on this line is 16ms long.
5. Blue Box: surfaceflinger grabs a view from the queue (Note in the yellow box that the buffer queue drops from 2 to 1.)
   a. Upon completion, this view is sent off to the GPU and the screen is drawn.
6. Purple Box: VSYNC-app now tells the app to render another view (and shows a 16ms timer)
7. Red box number 2: droid.yahoo.com measures the next screen draw.
   a. And the Cycle continues.

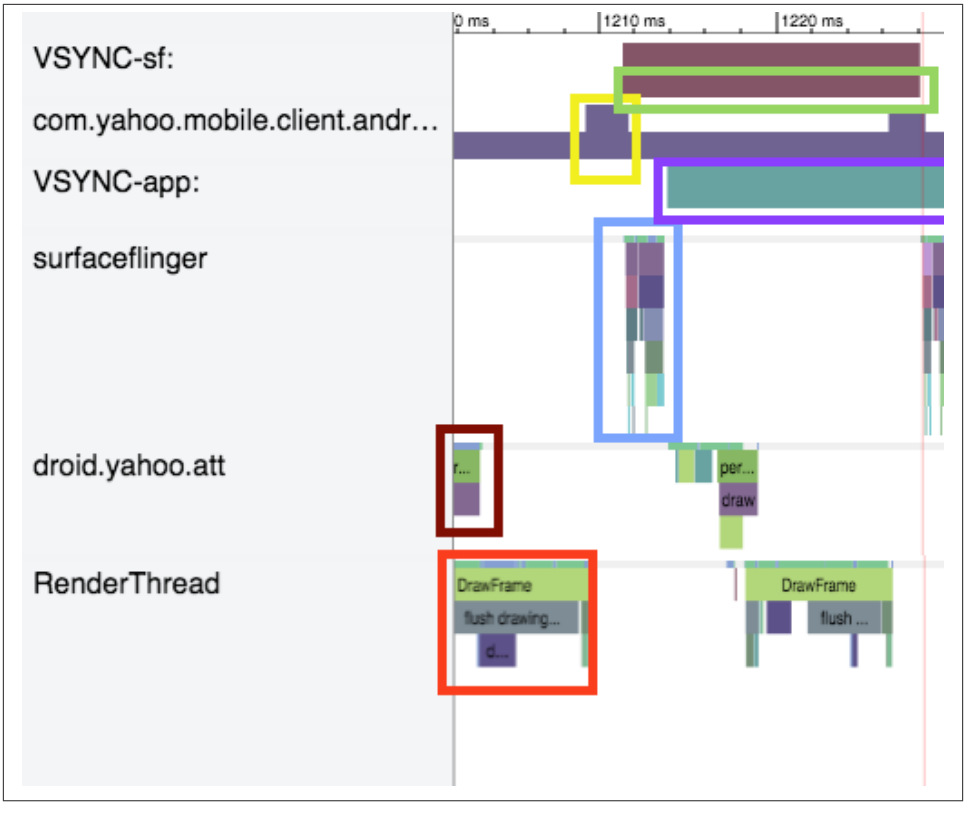


Figure 4-24. Systrace of Proper Rendering (Lollipop)

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.

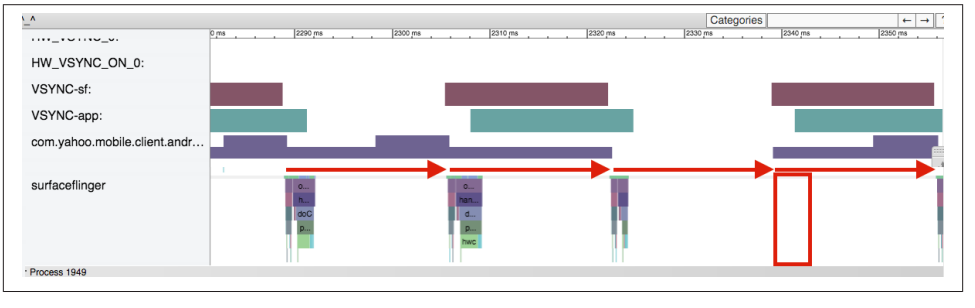


Figure 4-25. Systrace of Jank - Operating System View (Lollipop)

In Figure 4-25, we are looking at a closeup of the OS layer. To highlight the issue, I have added arrows indicating 16ms intervals, and a red box at the location of a missing surfaceFlinger. 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 1 and 2 views. As the surfaceflinger grabs one view (the buffer count drops), but it is quickly repopulated from the application. However, after the 3rd 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:

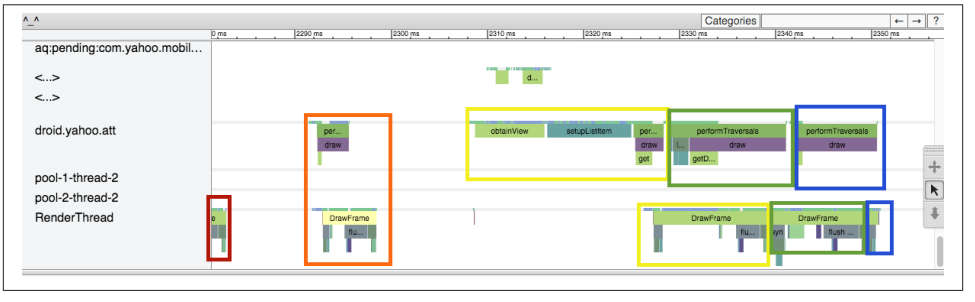


Figure 4-26. Systrace of Jank -App View (Lollipop)

In Figure 4-26, 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 7ms, (teal) "setupListItem" for 8.7ms, before the dark green "performTraversals" (3ms). The app then passes the data to the RenderThread, which is also significantly slower (12ms). Creating this frame took the app nearly 31ms (versus ~6ms 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.

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 is quick succession (green and blue boxes.) By quickly refilling the buffer queue, the application 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 following trace:

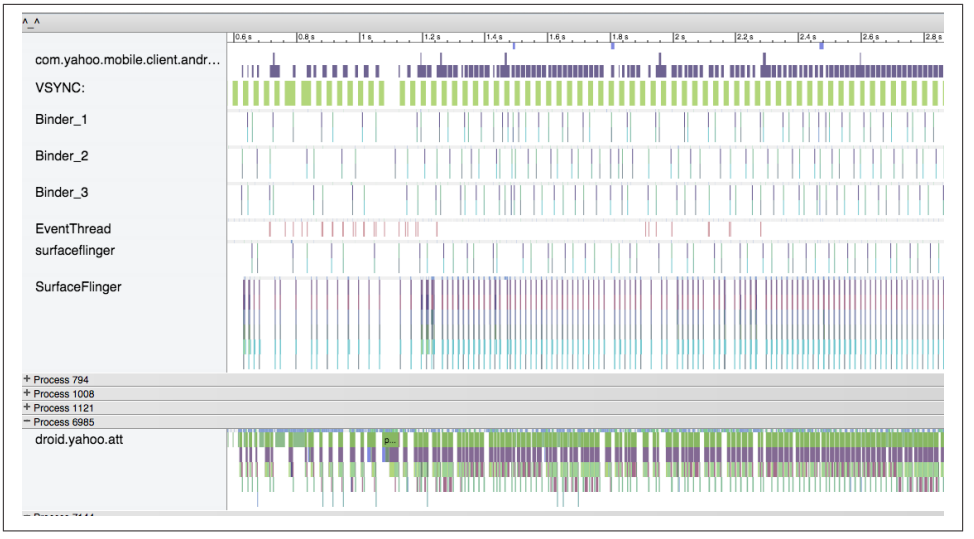


Figure 4-27. Systrace of Jank (Jelly Bean)

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 2 views in the buffer. On a device with a slower GPU processor, the application does not have as many opportunities to "catch up" and refill the buffer like the Nexus 6 did.

You can exceed the 16.6ms time to render a frame occasionally, since there are often 1-2 buffered frames ready to go. But, should you have 2 (or 3) slow frame renders in a row, your customers will experience jank.

Since this device was taken on a Jelly Bean handset, 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 application 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.

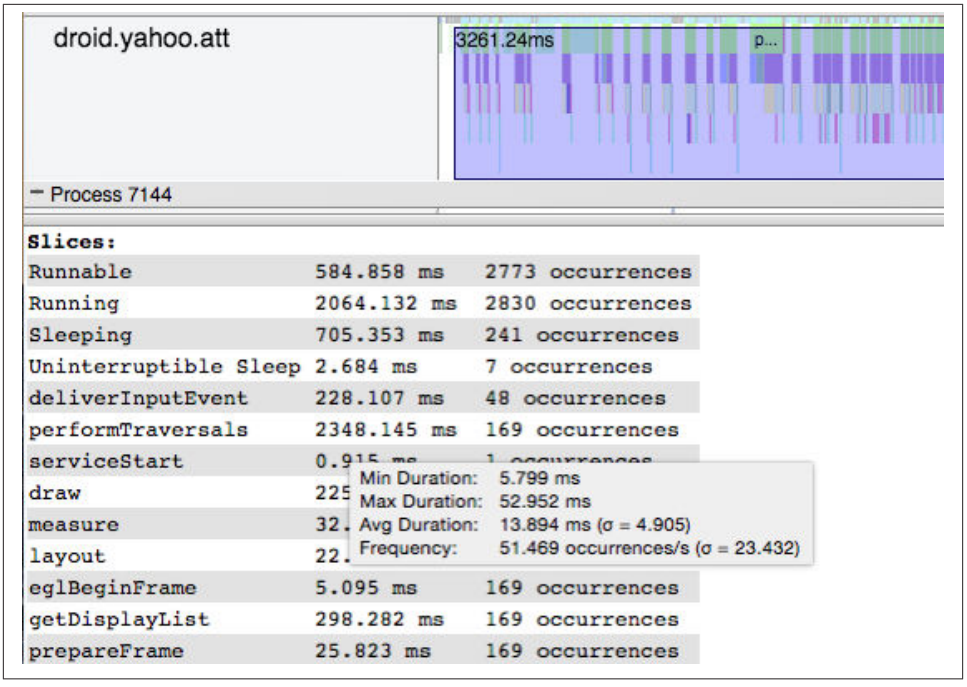


Figure 4-28. Systrace Data Summary

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. Here we see the "performTraversals" (the parent draw command) averages 13.8ms, with a standard deviation of 5ms. Since the 16ms jank threshold lies within one standard deviation of the mean, we can guess that there is a jank problem on this device. Zooming into this section Figure 4-29 shows this in detail. Each vertical red line indicates 16ms. There are five or six times instances here where the SurfaceFlinger misses the 16ms mark. The length of the green "performtraversals" lines are all nearly 16ms long (and since they must occur between each frame build, jank). There are also two blue-green deliverInputEvents (that take well over 16ms each) block the app from drawing the screen.

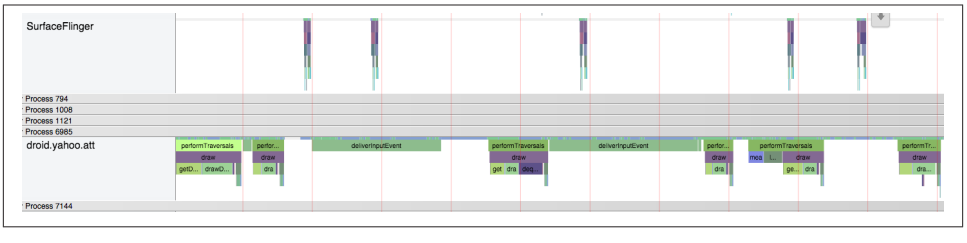


Figure 4-29. Systrace detail on a slower device(Jelly Bean)

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 application 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 application calculates a very large Fibonacci number (recursively) for each row of goat data. As you might imagine, this is very slow and CPU intensive. Since 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 4-20 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. Here is the "is it a goat?" app on an N6 using the unoptimized layout:

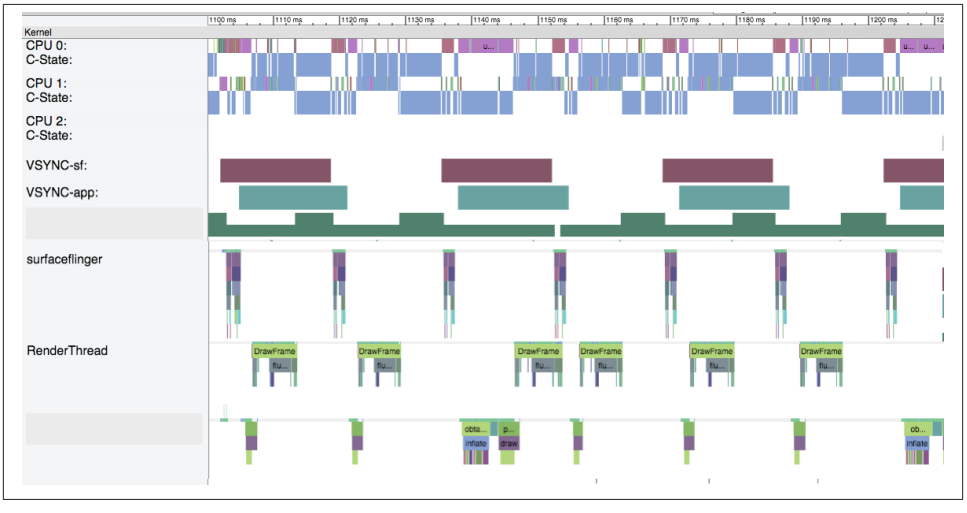


Figure 4-30. Systrace with CPU information

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 2 rows to the CPU reveals a neat pattern. When the RenderThread is drawing the layouts, CPU1 is running a blue activity (NOTE: 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 2 CPUs. Note that the major clicks on the x-axis are 10ms each, and none of these processes take longer than 2-4ms.

When we add the computationally intensive Fibonacci calculation into the draw, the Systrace takes a very different view:

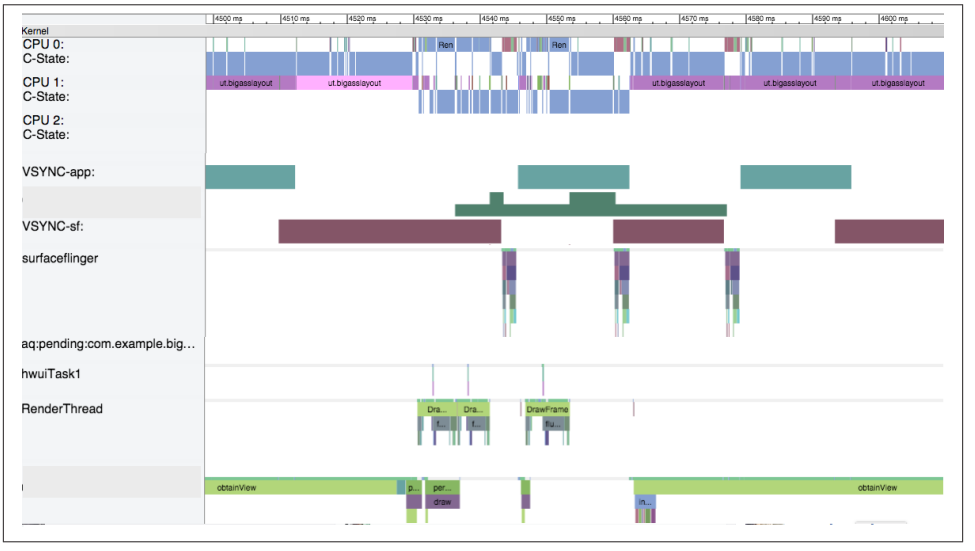


Figure 4-31. Systrace with CPU information and Delay in Rendering

This Systrace shows a lot of jank. In the same 100ms timeframe, only 3 surfaceflinger views are drawn (versus 7 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 2-4ms, but now range 2-17ms long. The large Fibonacci calculations are taking 13-17ms each, and this is really slowing down the application's ability to draw smoothly. We'll look at how to diagnose CPU performance (and its affect 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 4-26, I highlighted each frame as it was updated. In the new version of systrace, 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:

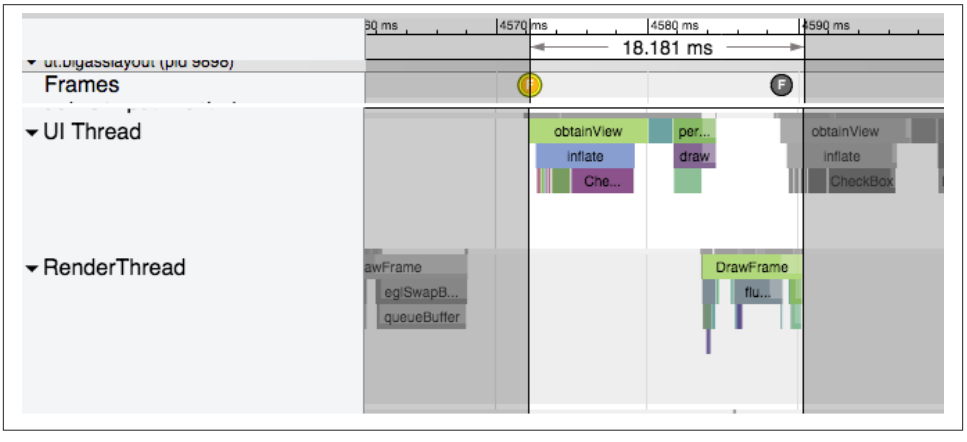


Figure 4-32. Systrace Update with Frame Highlighted

The new Systrace also has a lot better descriptions as to what is happening. In Figure 4-32 the frame render time is 18.181ms, and is colored yellow - as many frames in a row over 16ms could lead to jank. In the description panel below the trace, it warns that my application is recycling a ListView item, rather than creating a new one - and this is slowing down the view inflation:

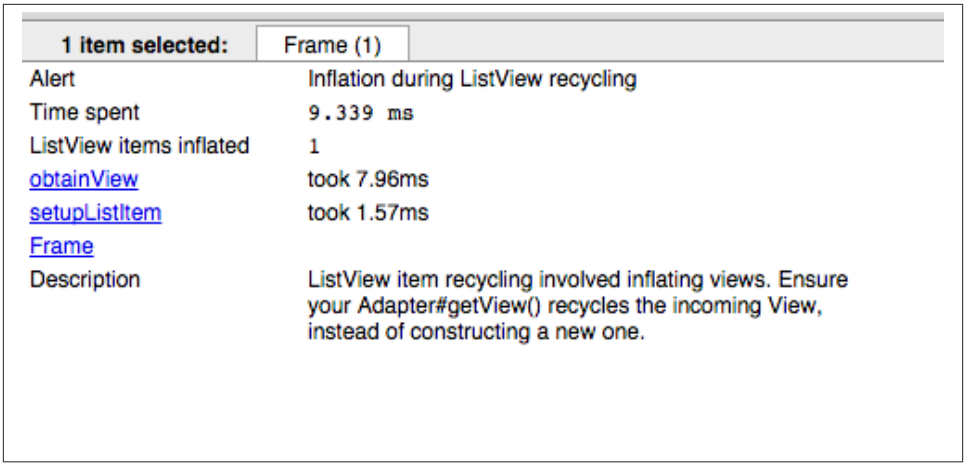


Figure 4-33. Systrace Update with Frame delay Information

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:

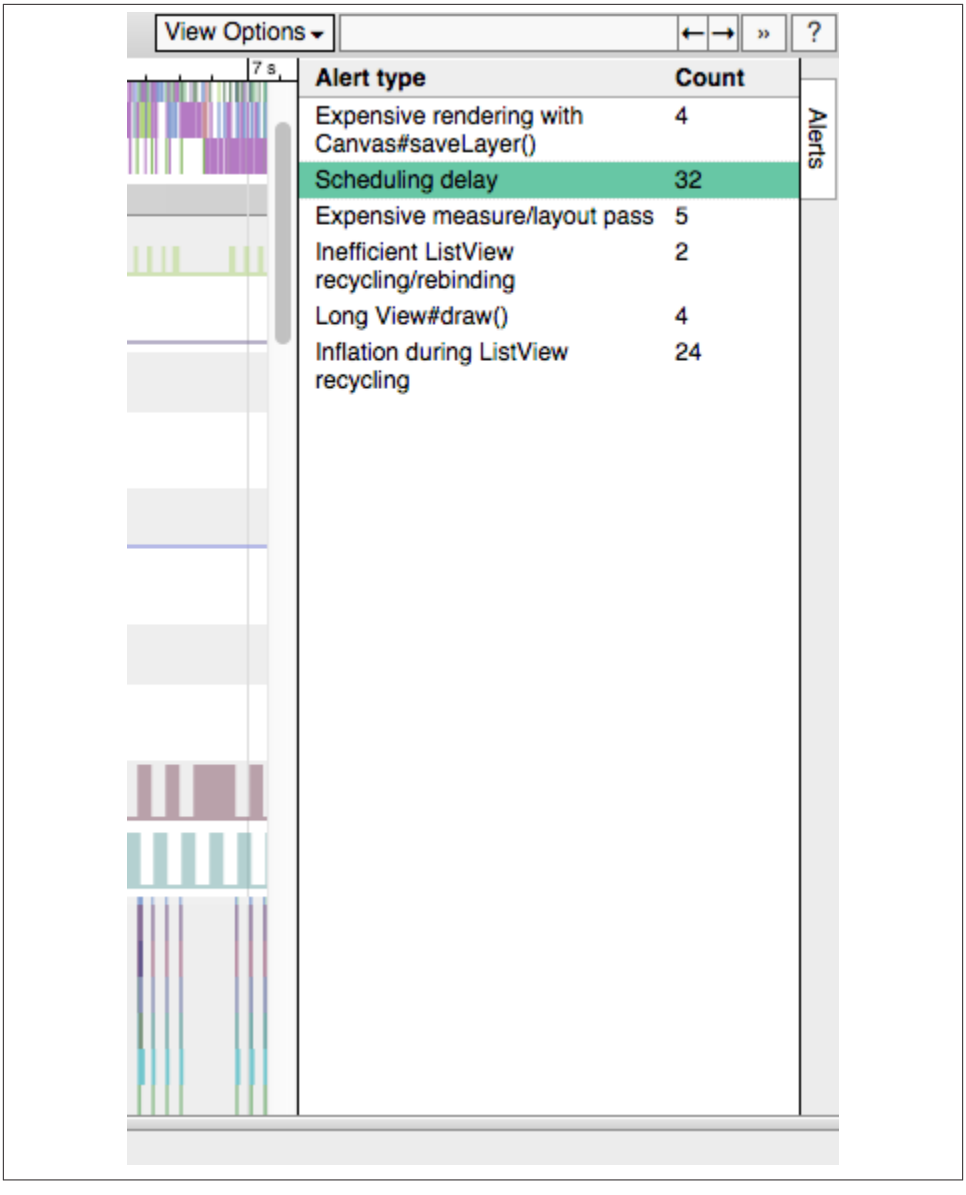


Figure 4-34. Systrace Update Alert panel

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 application 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

So, I just wrapped up a chapter on how to make your UI fast through testing, discovering issues and optimizing layouts. There is another possible way to make your Android UI faster: make it *appear* faster. Of course, it is crucial that you work work to optimize all of the code, views, overdraw and other issues that might affect your UI first - to really make your application as fast as it possibly can. Once you have done that, there are still a few ways to make your application appear faster to your customers.

The human mind behaves in interesting ways, and by changing the perception of waiting to the human mind you can make the delay appear to be less. This is exactly why grocery stores put trashy magazines in the checkout aisle, as having something to look at makes the delay less. 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 application.

Spinners: the Good and the Bad

Spinners/progress bars/hourglasses and other tools to indicate a pause have been around for years. They have also been used to make applications and transitions feel faster. When loading an application 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. T he developers of the iOS application 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

application 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 application 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 application had sped up (again no code other than the spinner was changed.) By changing the perception of the wait, the application 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 1 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 the 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 he server to update to update the UI. The mobile user does not feel the obligation 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 application appears to function when coverage is intermittent (like when your train enters a tunnel on the commute home). Flipboard has presented their queueing architecture that they use 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 applications 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 their app to never need a spinner.

Tips To Improve Perceived Performance

When the speed of your application 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. Since 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 application 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 walked through examples of view Hierarchy, and have 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 coming chapter, we'll look at how optimizing and reducing the CPU usage of your application will improve the performance of your application.

CHAPTER 5

Memory Performance

At then end of Chapter 4, we examined an issue where processes in the application blocked the UI thread, and preventing the screen to update. In Chapter 5, we'll look at how to measure and better understand how your application 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 2nd half of the chapter cover how to minimize the CPU usage of your application.

Android Memory: How it Works

Before we can discuss how to improve the memory efficiency of your Android application, 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 application by abstracting those details form your code, but there are important considerations to take while building your application 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 vs. Private Memory

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

Private memory is memory that is used just by your application and not used by other apps. Since 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 and 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 application starts, it is launched with a fork of the zygote process (giving your application 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 vs. Clean Memory

Dirty memory is memory that is only stored in he RAM, so if it were purged form the RAM - the application would need to be re-run 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 application 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. Since now the app code in memory is now by definition clean, memory management is ART is further improved.

Currently, since most devices are still using the Dalvik runtime, the most common type of memory is private dirty memory (memory only used by the one application, 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 re-used. In Figure 5-1, objects without active references (arrows) are colored in red, and will be removed when a garbage collection even occurs.

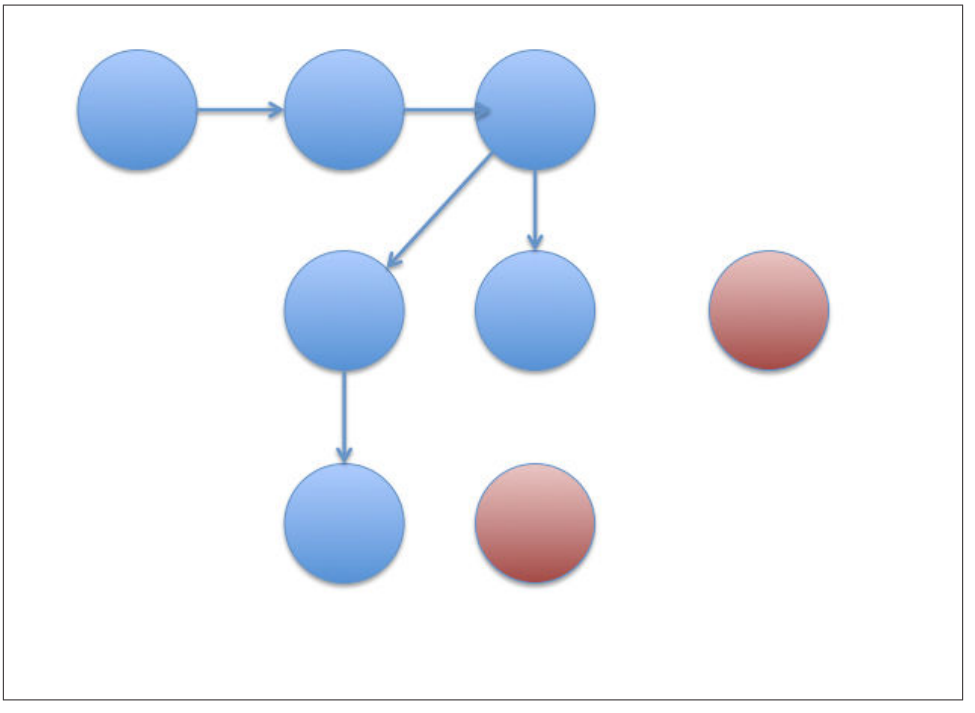


Figure 5-1. The Garbage Collector follows all references (arrows) marking active objects (blue), and collects all objects not currently referenced (red)

Garbage Collections Changes by OS

The GC in Android has evolved a great deal as Android has matured. Prior to Gingerbread, devices were low memory, so applications tended to have smaller heaps. The garbage collector was a "Stop the world" collector meaning that the GC cause 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 milliseconds, 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. While a partial GC does not clean up all objects that are unreferenced, since it does not travel the entire heap on each collection, it is faster. Instead of stopping your application from running, the concurrent GC runs alongside your app. This means that now there are 2 short pause times at the start and end of each GC, but they are generally under 5ms 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 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 1MB. This will be a problem if you are trying to create a 4MB bitmap - since you will get an Out of Memory error - there is not a 4 MB slice to 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 10ms 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).

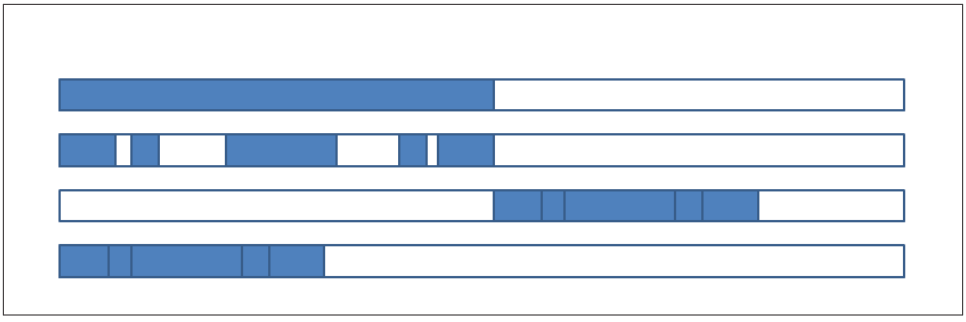


Figure 5-2. Garbage Collection a. Before Collection, b. mark and sweep GC releases small gaps of available RAM, c. semi space runs the GC, and rewrites to an open alloctaion d. compacting

There are a number of new GC algorithms in ART, but one interesting method that is carried out when an application is no longer in the foreground is the Semi-Space GC. Since the application is not in the foreground, rewriting the objects in memory is safe, and as seen in line 3 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 applications. When objects are moved in memory, the application has to be suspended to avoid errors. Since this can add to jank in an application, the semi space GC is only run with your application not in the foreground. This is not a fully compacting GC (see Note below), but it is a very useful way to open up large areas of unused memory.

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 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 not needing a new memory space, but rewriting 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 23.6ms. The app's heap is 50MB, is using 34MB leaving 31% free. The GC freed 5124 AllocSpace objects - relinquishing 199 KB, and cleaned up one Large Object from the Large Outpost Space of 16KB (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 application 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 application 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 why the reason behind GCs and memory leaks, so that you can 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 application, and how the system decides to clean up memory with garbage collections. 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 100x100 pixel thumbnail can use 320,000 bits of memory. Loading a number of these images at once, and you see how apps are using 50-100MB of your memory heap.

How much memory can you use on a device? ActivityManager.getMemoryClass will return the maximum size for your application'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 application, but it should be used with care as a large memory heap will actually slow down your application during garbage collection events (since the framework will have to hunt through more data for unused objects.) How do we find out how our application is using memory?

Running `adb shell dumpsys meminfo` on my Nexus 6 (with the "Is it a goat?" application 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)
 22116 kB: com.levelup.touiteur (pid 19038)
 19900 kB: com.google.process.location (pid 2917)
```

PSS stands for Proportional Set Size memory, and is the total memory used by your application. 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 141205 KB of the "is it a Goat?" app. Note that the PID for the goat app is 10821, 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 97?). The mediaserver 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 systemUI, NFC, and phone.

The next sections are where we can see the apps actually running on the device. In the Foreground is "Is it a Goat?" 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 Servcies and Cached applications 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.google.android.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.google.android.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.google.android.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.coffeestainstudios.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 vs. used RAM. In the case of my Nexus 6, it is clear that the apps cached in memory will likely stay in memory, as there is nearly 50% of the RAM 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?" application, while it is in the foreground on a Nexus 6. Let's break down what this table is telling us. We'll only worry about the data in the first two columns - the total memory in use by the application (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.

1. 13 MB from Dalvik (which I assume should read ART, but was not changed)
2. 66.7 MB is allocated to graphics
3. 26 MB dedicated to GL commands of rendering
4. 11.8 MB is Unknown
5. 6 6MB to "other"
6. 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 frag-

mentation 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, the number of activities. If for some reason, these numbers are much higher that you expect, wait for a second, and run the 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 application 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 to 6, 12 or 24 hours). 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.

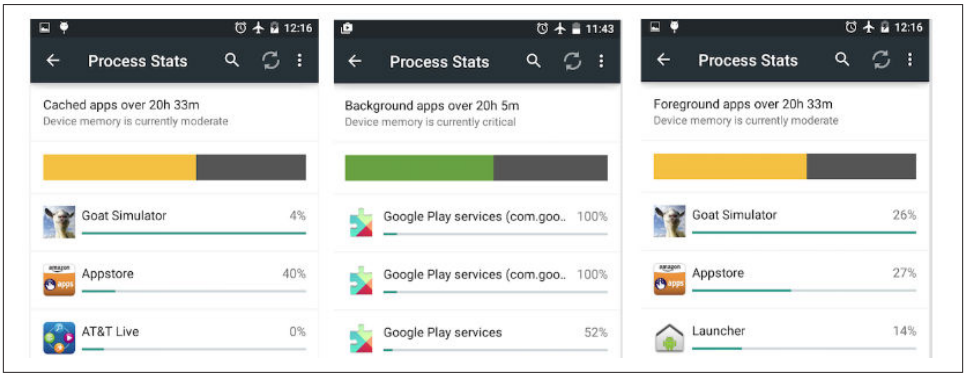


Figure 5-3. ProcStats Overview of app memory usage while cached (left), in the background (center) and in the foreground (right)

Each running application is listed with the % time it has been active, and the bar is a comparison of the average memory used by each application (again - you can see this for foreground, background or cached). Clicking on an application gives you detailed

information about how your application uses memory, and the RAM and run time in the state of the parent menu. To see how your app performed in another state (say Foreground to cached like in Figure 5-4), you must back to the main menu to change the state, and 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.

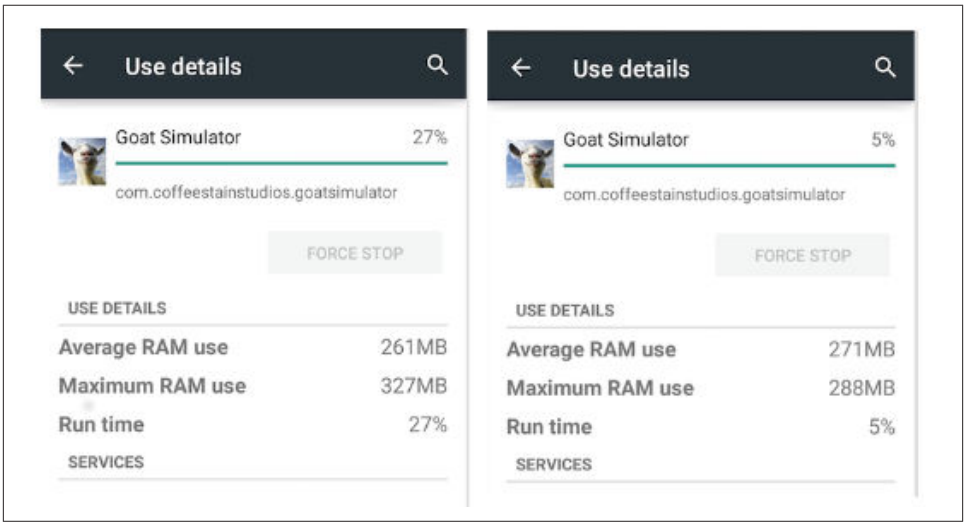


Figure 5-4. ProcStats For App: Foreground (L) and Cached ® - Lollipop

```
    Compare this to the wealth of information below. The dump contains the stats for the last 24 hours
```

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 304MB to 146MB 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? Looking at the Goat Simulator specific data in the report. 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 Procstats System Info tells us that with the screen off, the device was in a critical memory state for 1min 7sec. Looking at the Goat Simulator, it was active, or the last active app for 46+21s = 67 seconds. 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 application.

Below the timing, we get an addition memory usage breakdown of the app in the various states

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 in to 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 have looked at how "Memory Cleanup (Garbage Collection)" on page 117 works, and how to determine if your application is using excessive amounts to memory. The procstats 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 application, 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 application 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. The onTrimMemory will tell you where your application is in the cache, and how you can help remove memory to prevent your entire application from being removed. If your application is running, and there are memory problems, onTrimMemory will tell you how the memory situation on the device is:

- 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 the performance of your application.
- TRIM_MEMORY_UI_HIDDEN Your application was just moved off the screen, so this is a good time to release large UI resources. Now your application is on the list of cached applications. 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 3 levels:

- TRIM_MEMORY_BACKGROUND - you are on the list, but near the end.
- TRIM_MEMORY_MODERATE - in the middle of the kill list
- TRIM_MEMORY_COMPLETE - this is the "you're next to be killed" warning

In the Procstats App Info for Goat Simulator, you can see that when memory is critical, and the application goes from Screen on top to 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 application performance.

Even though memory in the Android runtime is managed, developers must sill worry about how memory is being used. Memory leaks are possible in Android applications when objects are necessarily 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 application (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 application actually use in memory? What sort of files are allocated into memory when your application runs? A great tool to better understand this information is the Heap Dump in Monitor DDMS. To activate the Heap Dump,

select your application, and enable the "Update Heap" button (its 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 application is doing, 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):

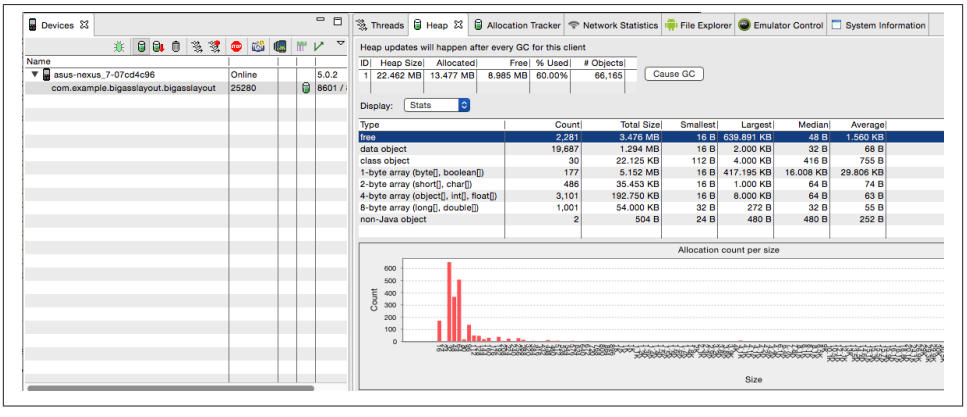


Figure 5-5. Heap Dump Results for Unoptimized App

The heap tool shows the device on the left, and in the center has a number of tables breaking down how memory is allocated. Below the table is a bar chart showing the number of objects by size. looking more closely at the table, we can learn about how memory is allocated in the application:

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 "Am I 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 is byte[] has the most memory allocated to it.

Another interesting feature of this is that the application keeps 40% of the heap size as free, but also keeps a large portion (nearly 3.5MB) 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 2281 free spaces, the smallest free space is 16B and the largest is 639KB. However, the median is 48B, meaning that half (or 1140) of these allocated free spaces are under 48B. 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 application. .

As you might recall from Chapter 4, the "is it a goat?" application has various view files that go from unoptimized to optimized (to come). Re-running 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? By Comparing Figure 5-6 with Figure 5-7, we can find out:

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 Comparison: unoptimized top, optimized bottom

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 1,254KB (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-byte and 8-byte arrays. The 1-byte arrays are unchanged at 5.152MB. Images are stored in 1byte 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, since 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 application 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 "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.

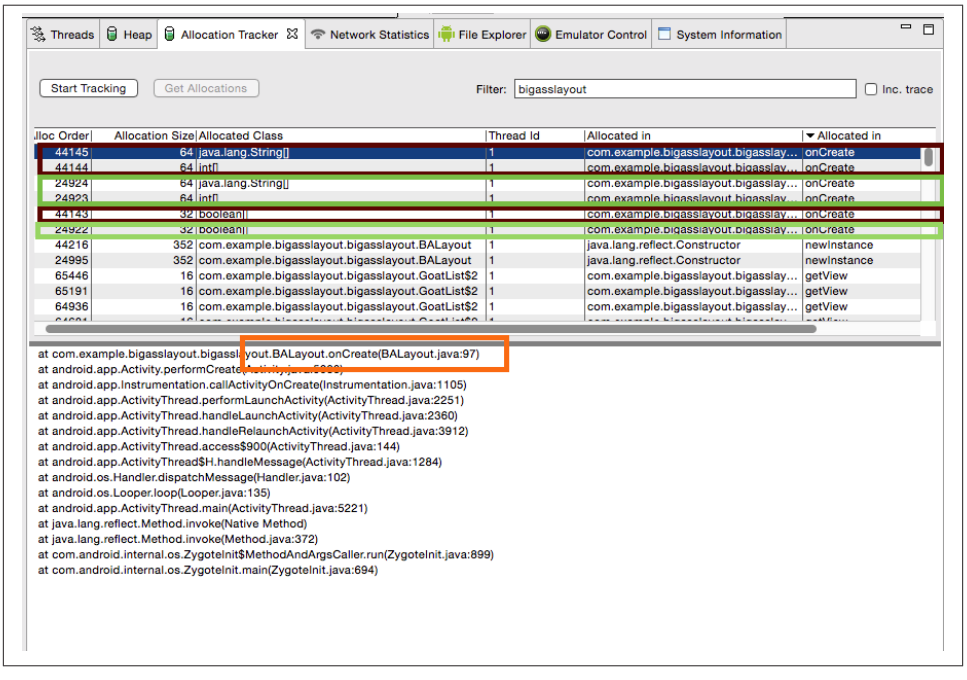


Figure 5-8. Allocation Tracker showing redundantly created arrays

In the test shown in Figure 5-8, I ran the "Is it a Goat?" application, 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. Since 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 3 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 bytes 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 3 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?" application, you can replicate this reports by selecting the "Create Objects During Render" box in the settings menu. This removes the arrays from the saved configuration, forcing the application to recreate 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 3 files recreated on every screen rotation.

Adding a Memory Leak

I have added an option in the "Is it a Goat?" application 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 above is not a huge one. In order to radically inflate the memory heap of the application and see an out on memory error, the Iceberg.sink object creates a 2MB Byte array (mostlyUnderwater) and adds it the the ArrayList iceSheet. On devices with lower available memory (in this case an samsung 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 above logs show the memory changes to the application when I rotated the screen twice. The heap grows by 2MB (02:10:31.680 and 2:10:35.445) after each rotation by 2 MB (to 19Mb and then to 21 MB). There are 4 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 application, as they pause the system for 36, 18 and 24ms each. The GC_CONCURRENT is the general

GC that runs periodically to clean up objects, and its 11ms 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 below we are not at 58.5MB), 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)
```

The garbage collections after the heap grows to 58.5MB shows what Android does to prevent a out of memory crash to your app. The application is attempting to allocate another 2097168 byte 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 2MB segment of memory for my byte array, the application 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 Leak Canary

In order to diagnose where your application is leaking memory, you will need to analyze all of the files that your application 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 apps 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, it is run automatically.

When creating your heap dump, try to replicate the steps that cause large memory issues. If you can get your application 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 application while you are debugging.

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 application 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. Since 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:

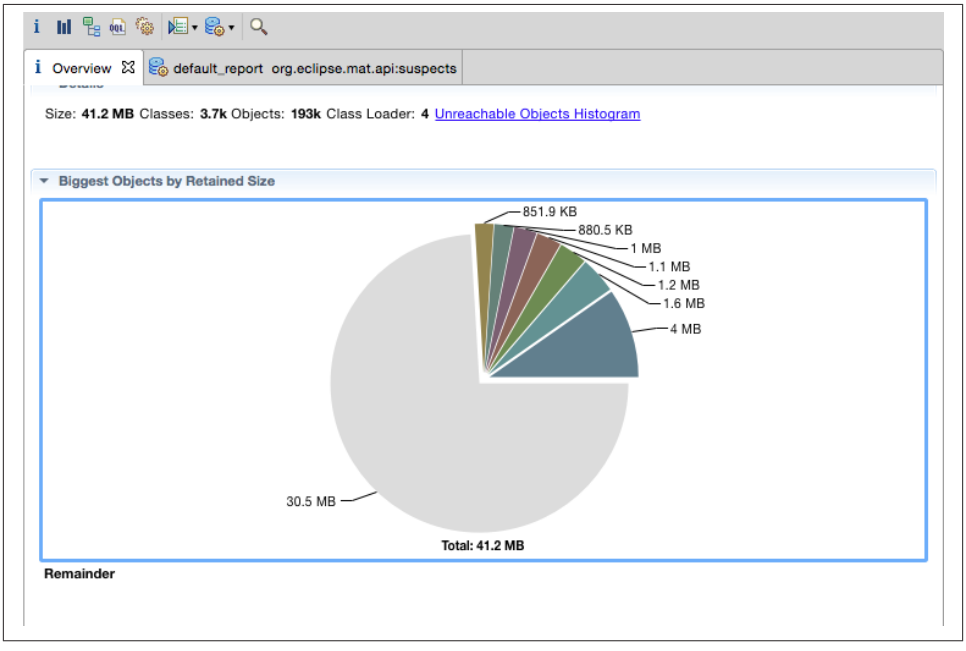


Figure 5-9. MAT Overview

The MAT tool provides a wealth of data in a number of different windows. In figure-MAT_Overview, 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. In the shot above, the largest chunk of memory is gray, and represents free memory. The 2nd largest class is the Iceberg class (as a result of the large ArrayList holding 2 byte arrays) weighing in at 4MB.

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.

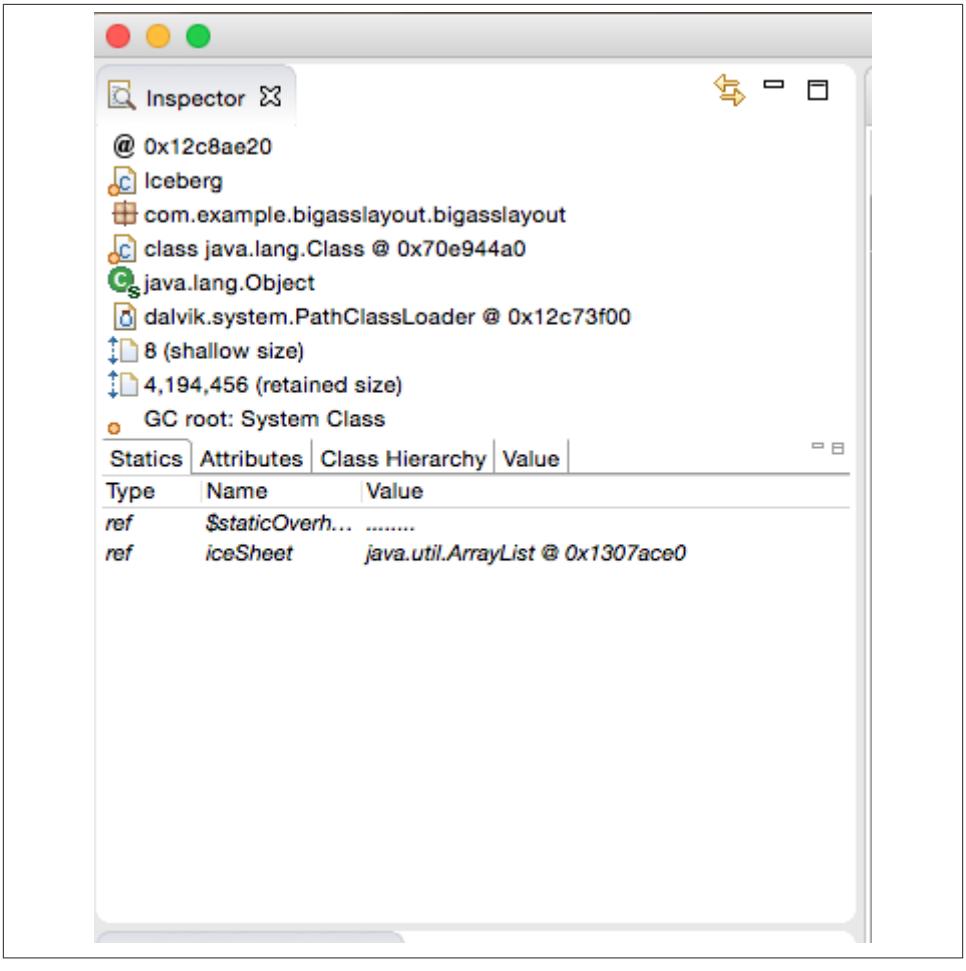


Figure 5-10. MAT Inspector Window

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 left graph which was run after only 2 screen rotations, there are 2 suspects indicated, one using 27 MB (byte arrays) and the other at 6.1 MB (java classes). The chart on the right was run after many screen rotations, and the memory utilized by byte arrays remain at 27 MB, but the java class memory allocation has ballooned to 36 MB. If we had not known already, this looks like a good place to find a memory leak.

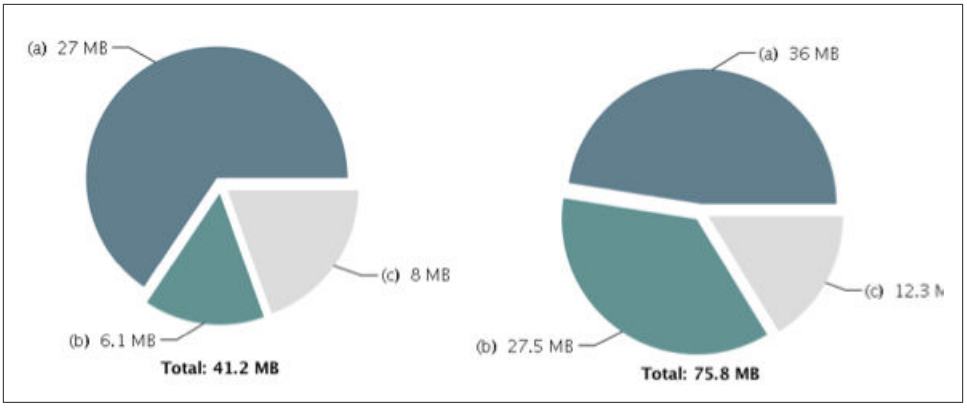


Figure 5-11. MAT Leak Suspects from 2 Memory Heap Dumps

Below the pie chart are yellow boxes describing all of the suspects. In this case, we'll continue our analysis with 2nd trace (run after many screen rotations.)

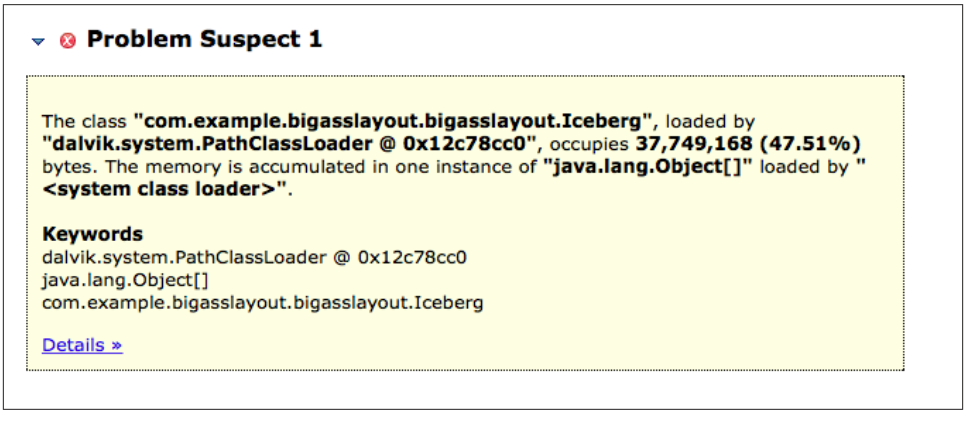


Figure 5-12. MAT Leak Suspect 1

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.

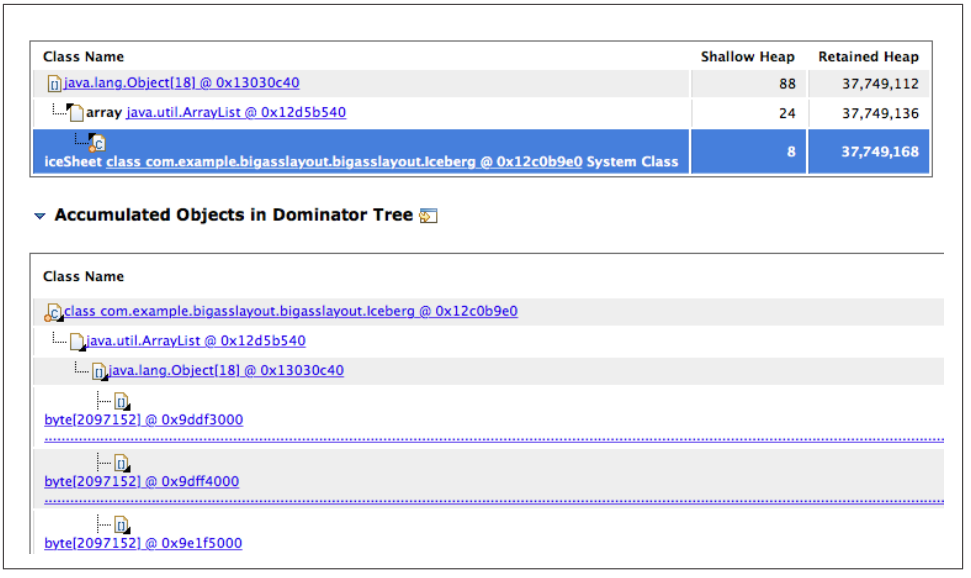
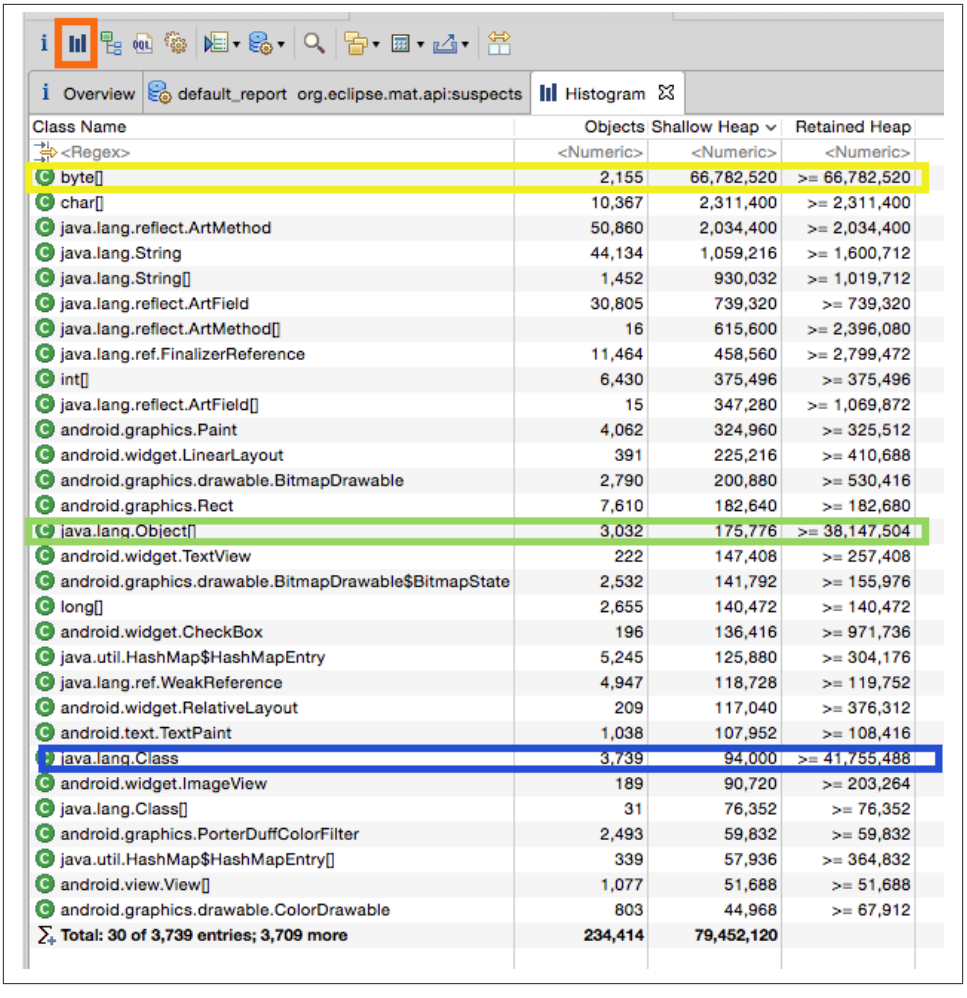


Figure 5-13. MAT Leak View

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 8B, but a retained heap of 37MB. 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 2.09MB 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.



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

A memory Histogram is created by pressing the icon that looks like a bar chart (marked by an ornage rectangle in Figure 5-14). 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). 66MB is more that I would expect here, especially since Figure 5-11 shows just 27 MB of byte arrays as images. * java.lang.Object[] (green box) has a low shallow heap, but > 38MB 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 cannout find your activity (or a class you are interested in) in the Histogram view, clicking the <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, shown below (and sorted by retained heap):

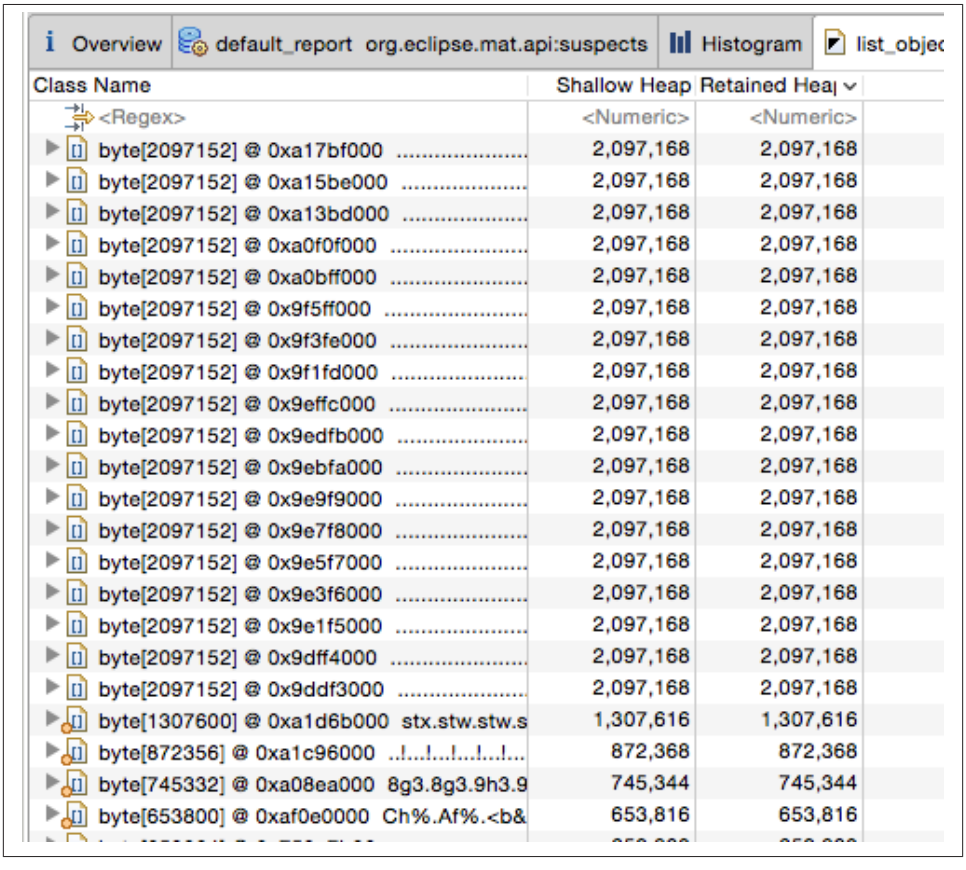


Figure 5-15. MAT Byte[] Objects

At the top of the list, we can see the 18 2MB 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:

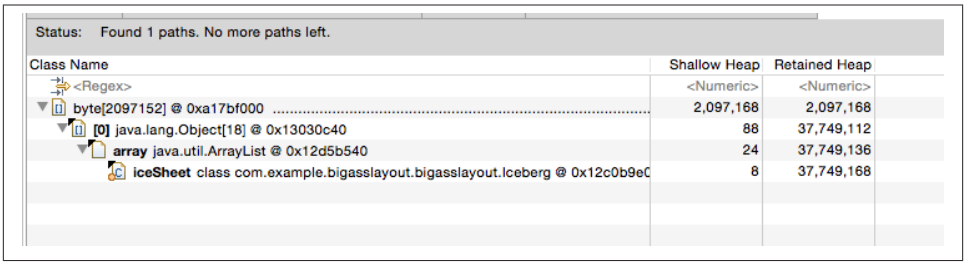


Figure 5-16. GC roots of 2 Byte[] Objects

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 repot 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 High Performance Android repository Github. 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 application allocates memory is a fascinating way to learn about how Android handles memory allocations, and find ways to optimize how your application 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, an open source test tool that automates a lot of what MAT does.

LeakCanary

LeakCanary was developed at Square to reduce the number of OutOfMemory errors that they were encountering with their application. 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. Since this

approach was slow, they wanted to find the memory leak in their code, before launching to their end users. LeakCanary was born. Since it sniffs out memory leaks before any OutOfMemory crashes, it is the "canary in the coal mine" for memory leaks. Since using LeakCanary, Square reports a 94% drop in OOM crashes! Let's see how this tool works!

It is super easy to get LeakCanary up and running. It is implemented in the AmIaGoat app on GitHub, but the instructions on Github are clear:

In the `build.gradle`, add 2 dependancies:

```
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 AmIAGoat, 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:

```
//leak canary 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 "Am I a Goat?" app, and 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 (154ms for the GC, 2 seconds to collect the heap dump, and 14s to analyze), and what object in the class caused the leak. The Github documentation walks through the steps to report the leak and the heapdump 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!)

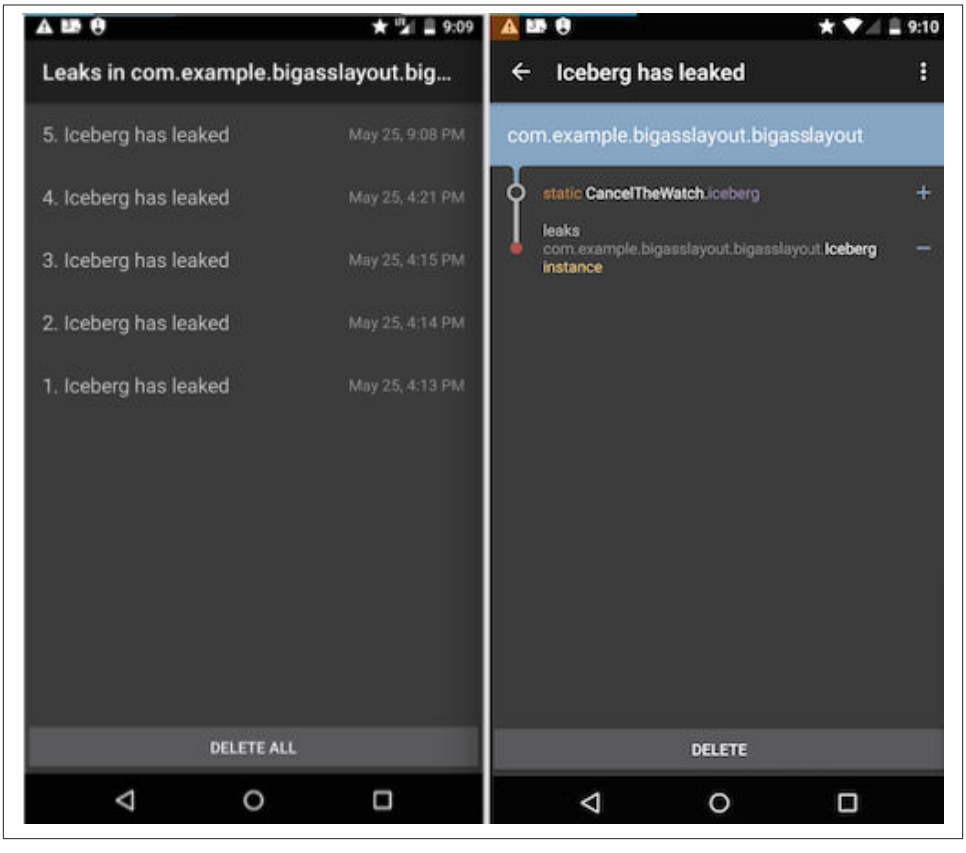


Figure 5-17. LeakCanary Screenshots: Summary (L) and Detail [8]

Finally, the reports are also shown on your device in the notification bar, and in a new app in your app list called "Leaks." Leak Canary will store the first 7 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 application, reducing the number of crashes, and improving your app's performance.

Memory Summary

Until very recently, the only way to discover memory leak issues was to study all of your OutOfMemory 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 application handles memory operations, your application will run more efficiently on memory constrained devices, and the number of outOfMemory 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 application - keeping the GC from blocking the main thread of your application. 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

As we continue our journey to high performance Android Apps, we've looked at battery, UI, and memory management performance, and how optimiing how these function in your app will reduce crashes, and speed up the performance of your app. In this chapter, we'll cover an essential part of the Android device, the CPU. The CPU is the *brain* of the device, and since the CPU processes all of your code to create your application, 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 in to 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 interchanged for any computations), 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 e-mail), a lower powered, more efficient CPU will be tasked with the job. When a video or a game is being played, 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 ca 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 application 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 application, and how to determine what threads or processes in your application are causing strain on the CPU. We'll look at how improper use of the CPU can block ren-

dering, or even cause a dreaded "Android is 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 ten application 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* 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.

Now, 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 /
                                                                                fa
  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 applications (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, the CPUinfo is also available as an overlay on your device through the Developer options. 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.

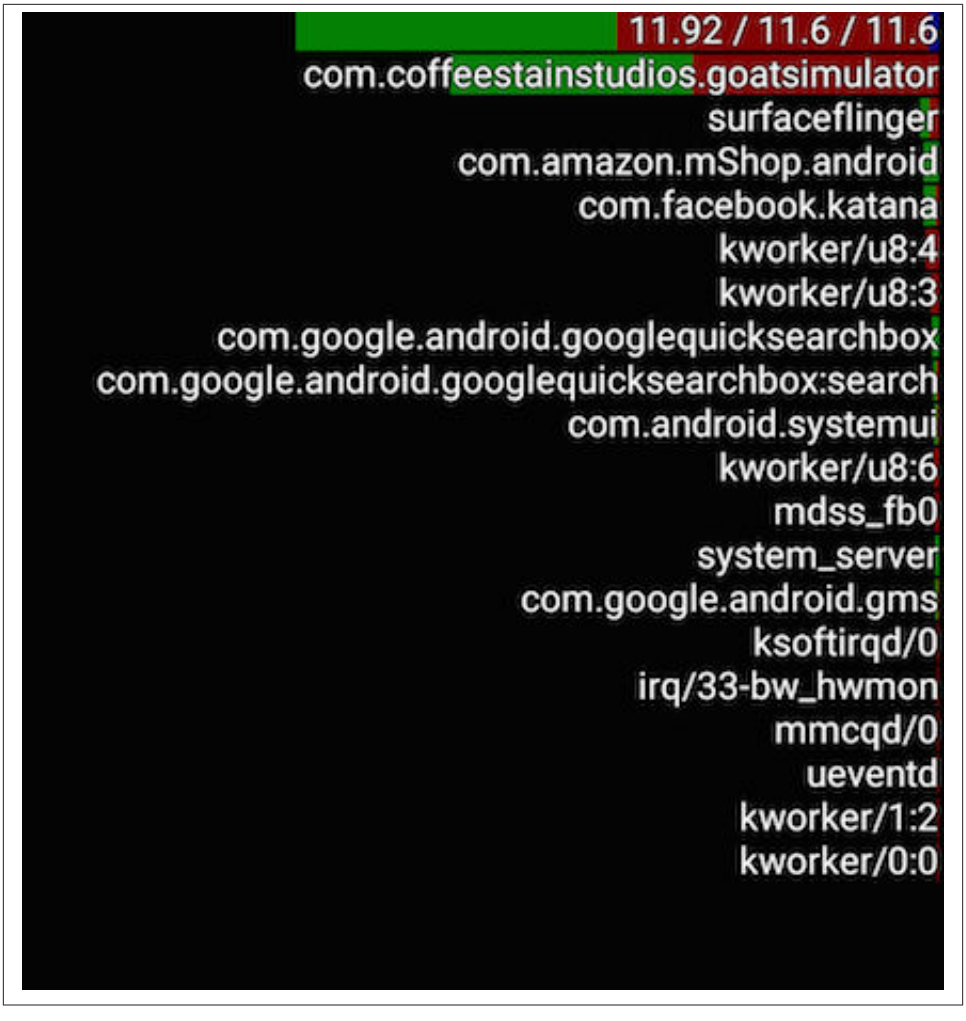


Figure 6-1. CPU info Overlay

Systrace for CPU Analysis

While top and cpuinfo provide basic understanding on memory usage of your application, we still need to dig deeper into the CPU cores to see what is actually running on the CPU cores while your application is running. In Chapter 3, we looked at the "Systrace and CPU Usage Blocking Render" on page 107 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. You can look at the traces described in this section, they are in the book GitHub repository (trace4 has no jank, and trace7 has jank.)

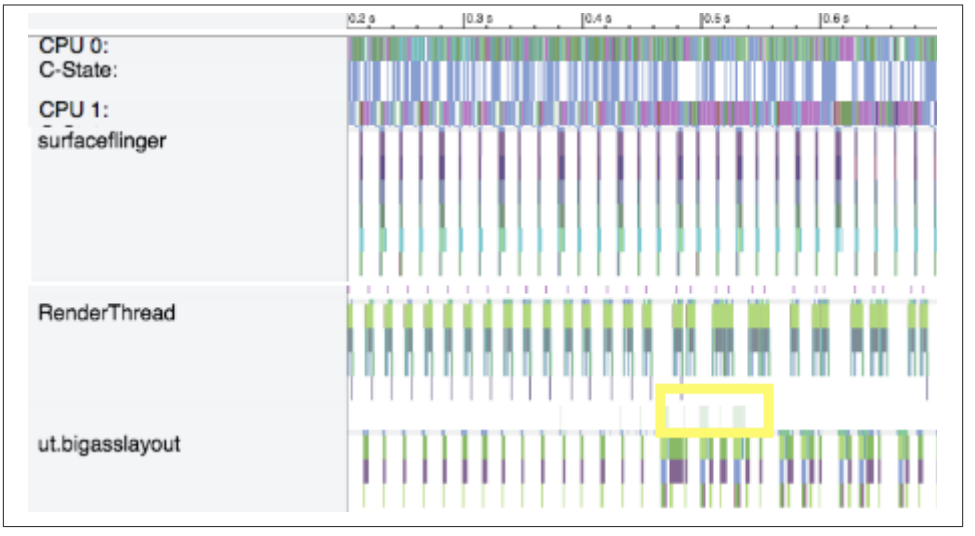


Figure 6-2. Systrace with no Jank

At the top of each Systrace (using the same setup as in Figure 4-21), are rows with information pertaining to each CPU on your test device. When I run a systrace with the regular views in "Am I a Goat?" 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 16ms as we expect. In the 2 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:

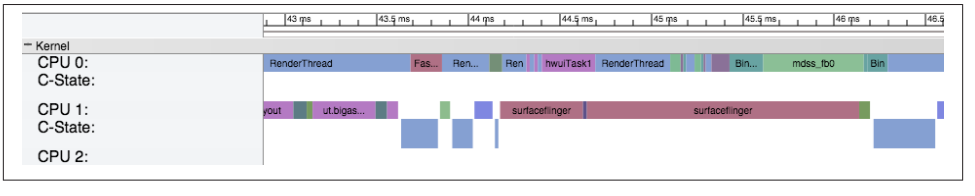


Figure 6-3. CPU view of Systrace

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

- purple is the AmIaGoat app
- blue is the RenderThread

- Brick red is the surfaceFlinger
- Many other extremely short processes.

If you look at (to come) 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. Inside the yellow box there are very light gray markers (they are tough to see in the tool, much less a screenshot) indicating that a user input has taken place.

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 (to come), but fewer views are rendered. For the first hundred milliseconds, everything looks great, no jank and everyone is happy. But about 150ms in, the UI gets stuck behind the calculation of a 8 digit fibonacci number. CPU0 (and later CPU2) go solid magenta, as the Am I 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.

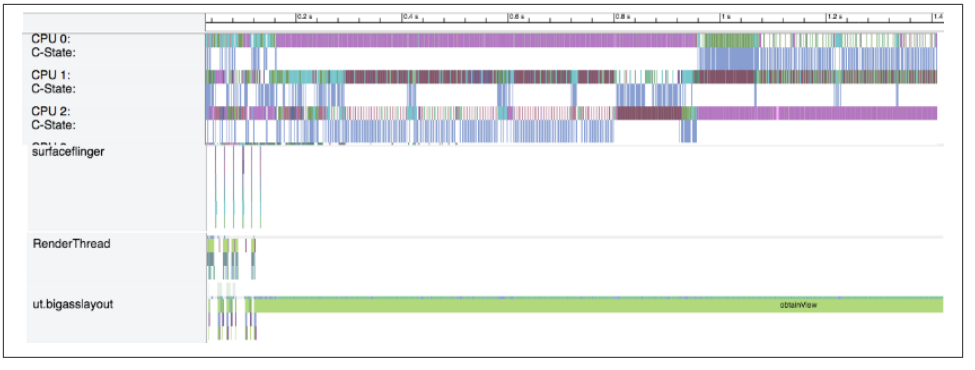


Figure 6-4. Systrace with Jank

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 15ms, and the thin lines abone the green obtainView (bottom row) are dark green and blue indicators. These indicators are telling you what state the CPU is in for your application. 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.

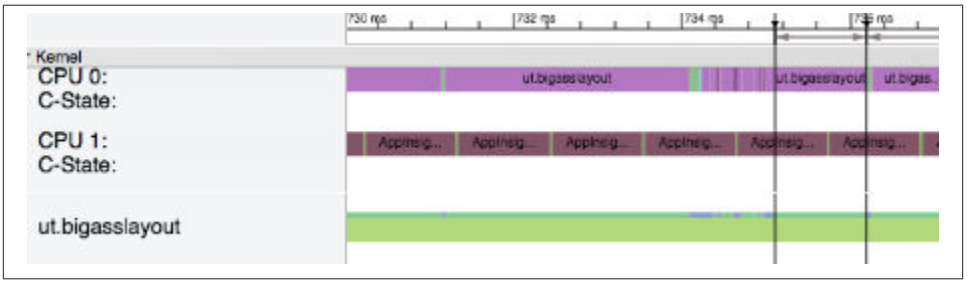


Figure 6-5. App CPU State in Systrace

With the knowledge that there is a process blocking your rendering, now we can apply the Traceview application to further diagnose the problem. There are two incarnations of Traceview, and both display the same information differently. Its worthwhile to discuss both tools, since 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 application, 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 microseconds (defaulting to 1,000µs). 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 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.)

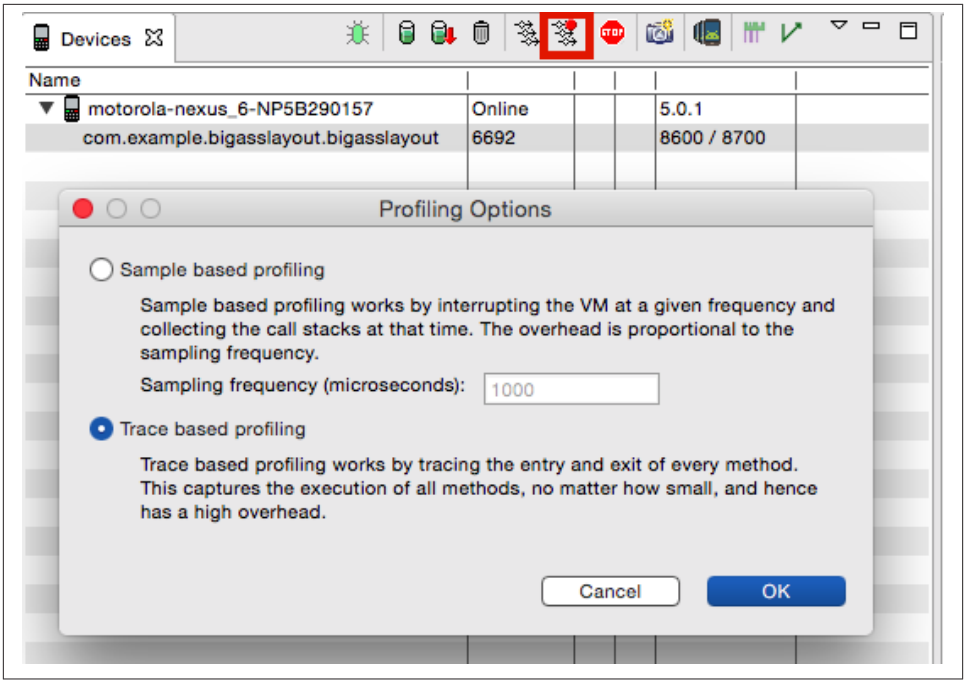


Figure 6-6. Starting Traceview

One you have started traceview, run the operations in your application 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 Am I a Goat, there is just the main thread). Each method is shown in is different color (and fully zoomed out — the start and stops all appear black.)

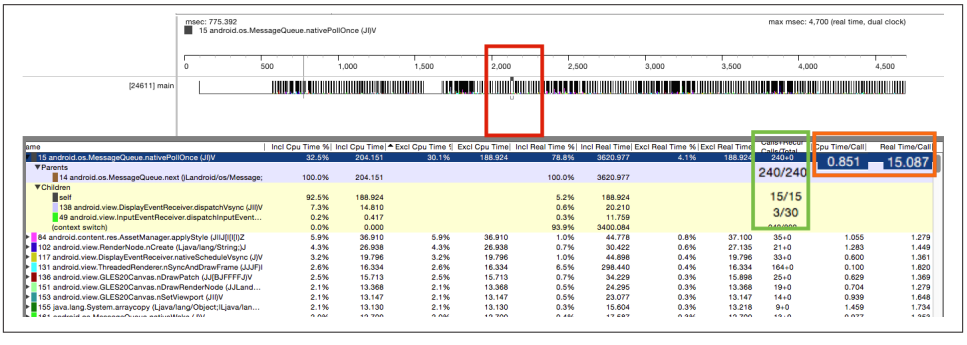


Figure 6-7. TraceView Overview

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 Am I a Goat application running in a normal manner (no issues). I have highlighted method 15 (android.os.MessageQueue.nativePollOnce) to show that it has the parent Message-Queue.next, and 2 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 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
- 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 15 times (and is the only parent of this method.) It calls method 49 3 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 (since 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 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 ~2,100ms in Figure 6-7, you can see one such call in the red box. Traceview denoted the method by adding a solid dark grey 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 < 16ms to complete. Scrolling in to a 500ms range, we can see that this method is called for each frame render (Figure 6-8 highlights every instance of method 15.)

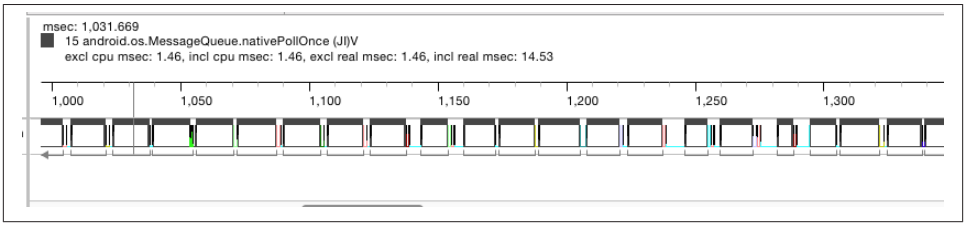


Figure 6-8. TraceView Zoomed In Showing a highlighted method that Reoccurs Every 17ms

Now that we have seen how an app should behave in traceview, let's look at a Traceview of Am I a Goat with the Fibonacci counter turned on. Immediately, we can see a difference:

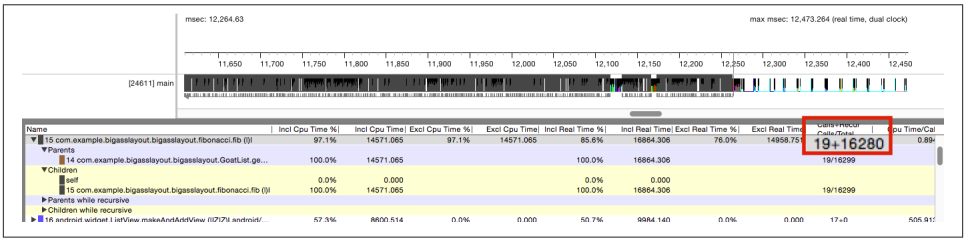


Figure 6-9. TraceView of Am I a Goat with Fibonacci Calculation

From 11,600s to 12,250s, 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 application. From 12,250 - 12,450s, we return to what we would like to see - regular 16ms 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 since it is a recursive statement, we see that it calls itself another 16,299 times during the trace (enlarged text in red box). The entire trace is over 19s, and nearly 17s is 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)

Since 0.2.10 of Android Studio, a new Traceview was released, 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 methods 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:

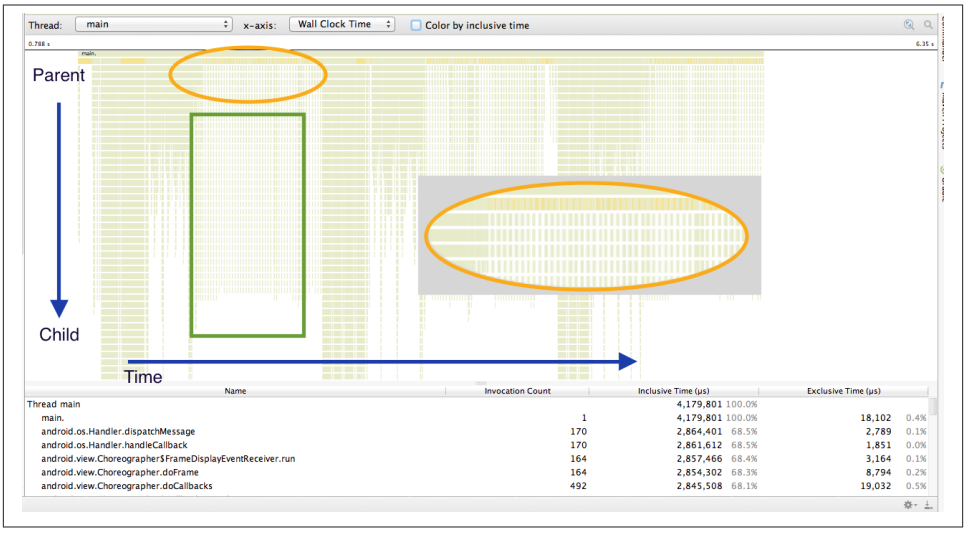


Figure 6-10. Flamchart Traceview Overview

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- green based on the amount of time each takes. By default, this is set to the exclusive time, and there is a high level method (in the 2nd 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 dependancies (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 method GoatList draws each row in Am I a Goat? 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).

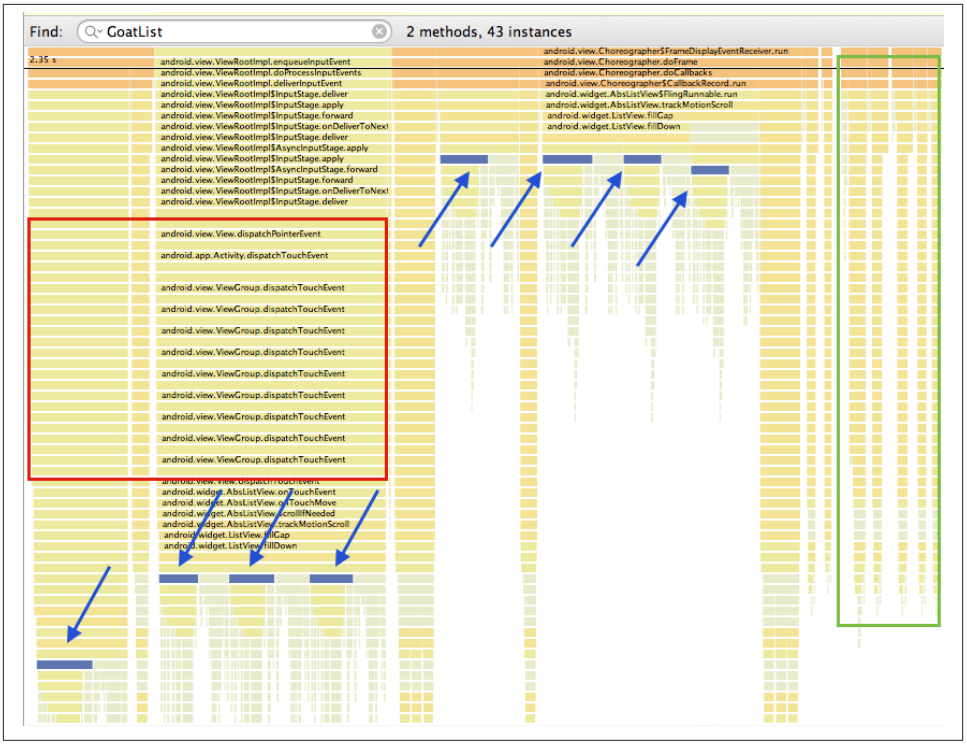


Figure 6-11. AS Traceview GoatList Filter

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 optimied 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 (26ms vs 40ms.) 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 (orang box), as it has two possible states and an animation when the states toggle.

Figure 6-12. AS 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 5-6ms is not present. In the app, there are 10 checked rows (they are goats), and 3 that are unchecked (not goats). When I originally wrote the app, I had all 13 boxes default to checked, and then unchecked the 3 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:

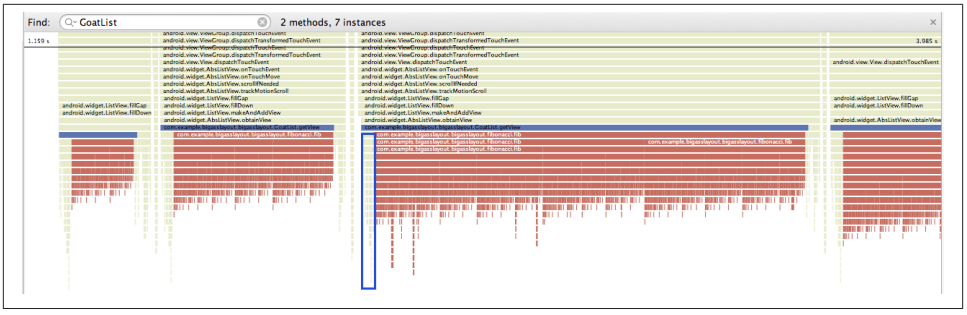


Figure 6-13. AS Traceview GoatList Fibonacci delay

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 (and 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 are denoted in red as Traceview recognizes that these calculations are causing a slowdown in the application).

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 place overlays on your device showing memory, CPU, battery, network and other characteristics as an overlay on your screen while you test your application. If your phone is 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 db 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.

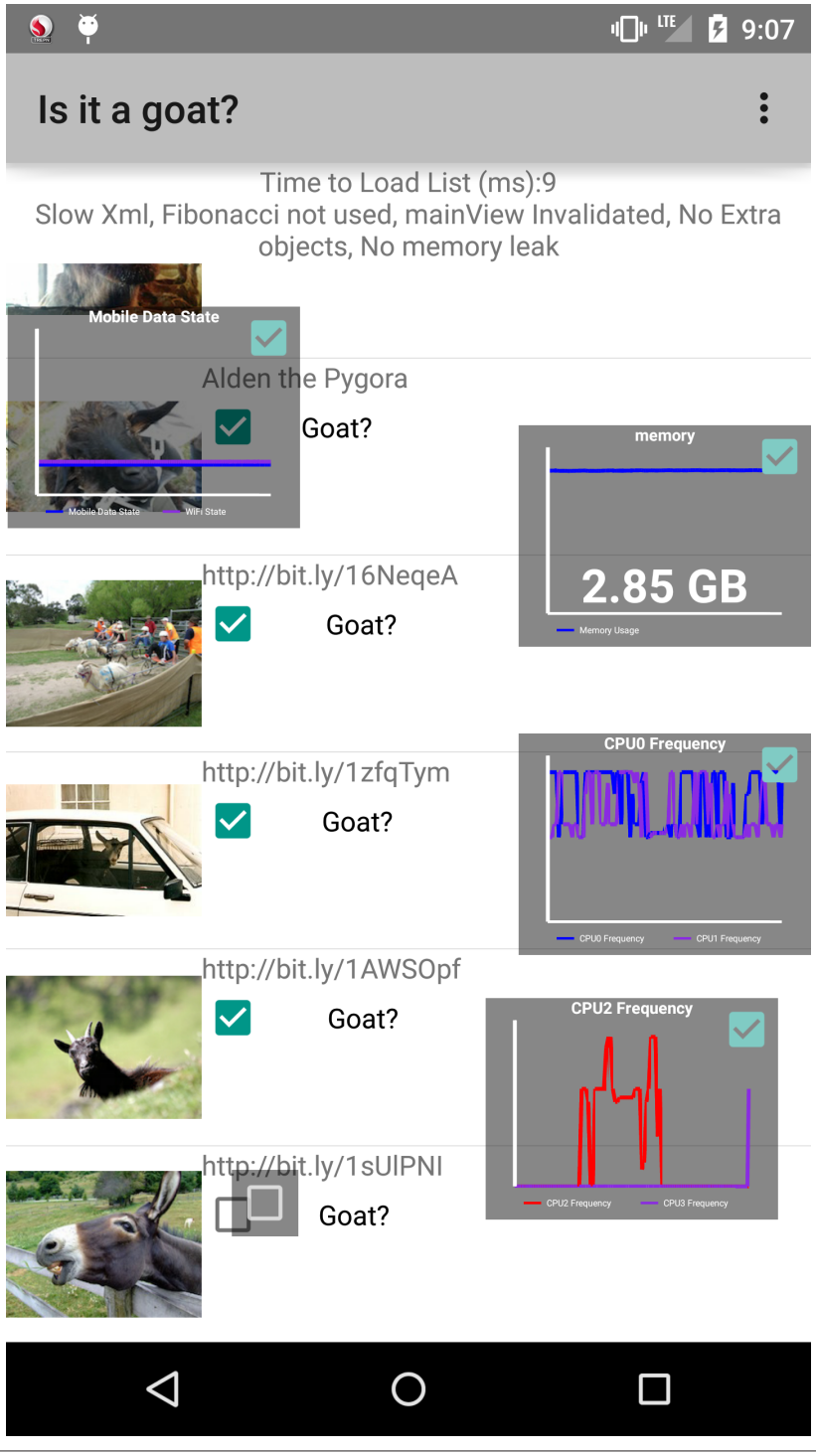


Figure 6-14. Trepn profiling CPU in Overlay

This is another great way to visually see the CPU usage of your app.

Conclusion

The tools described in this chapter are all free to use and provide a great deal of information to developers working to debug issues in their apps relating to memory and the CPU. Reducing the CPU footprint of your application allows the surfaceflinger and framebuffers to ensure 16ms frame updates and a jank free experience. Reducing the CPU footprint of your app will also save memory, battery, speeding up the application and reducing 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 applications use existing networks, and how important network usage affects performance of your application but also of the device's battery. As we determined in (to come), the cellular, Wi-Fi and Bluetooth radios that facilitate all of this amazing communication are also major factors in battery drain. By maximizing your application's network performance, you can make your applications 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 application for different network environments (since much of the world has only 2G and 3G coverage, you should ensure your application 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 vs. Cellular Radios

Wi-Fi vs. 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 application 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 2 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" on page 27 table, that 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 the above description is that you are seldom in ideal Wi-Fi conditions. Since 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 240ms to turn off.) As we saw in the "Android Power Profile" on page 27 section 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 application 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.

Table 7-1. Evolution of Wireless Generations

Name	Gen.	Down Max (Kbps)	Latency (ms)
GPRS	2G	237	300-1000
EDGE	2G	384	300-1000
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 the "Android Power Profile" on page 27, 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 125mA) uses less power than Wi-Fi (at 240mA) 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 State Machine that controls how data connections are established and taken down.

State Machines

A State Machine (sometimes a finite-state machine) describes a logic sequence of events with a finite number of states. A simple state machine is a light switch with states off and on. In thes 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 and 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 500-1,000ms latency to the time establishing a radio connection. This latency and delay is one of the crucial aspects of mobile connectivity that the cellular Radio Resource Control (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 balancing power consumption. Each carrier can specify various states, and the time a device re-

mains in the various states (and thus, each network is slightly different). Since each network around the world has different specific variables, the exact timings are not important to know, but a understanding the basics of the RRC State Machine is important to understand 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 10-15s. 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.

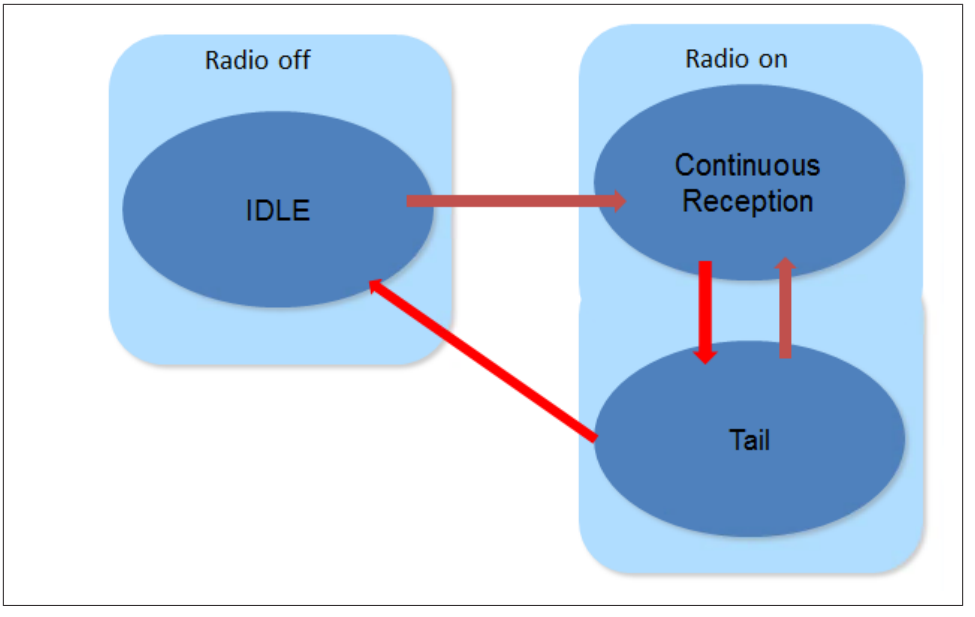


Figure 7-1. The LTE RRC State Machine

In 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), which reduced the latency to establish a radio connection by a factor of 5-10 from 3G networks. What can be 300-1000ms on 3G is now 50-100ms 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 (and covers all network performance is 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 (since they are so small). So, generally speaking, downloading content over LTE will use slightly more power than on 3G.

Radio Connection v. Data Connection

There is a nuance that exists between the radio connection and the data connection that bears discussion. 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 applications. All data connections cost at least 10s 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 application 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 59, we looked at a study where 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. When the call ends, the radio is able to transition back to the LTE (blue) data network.

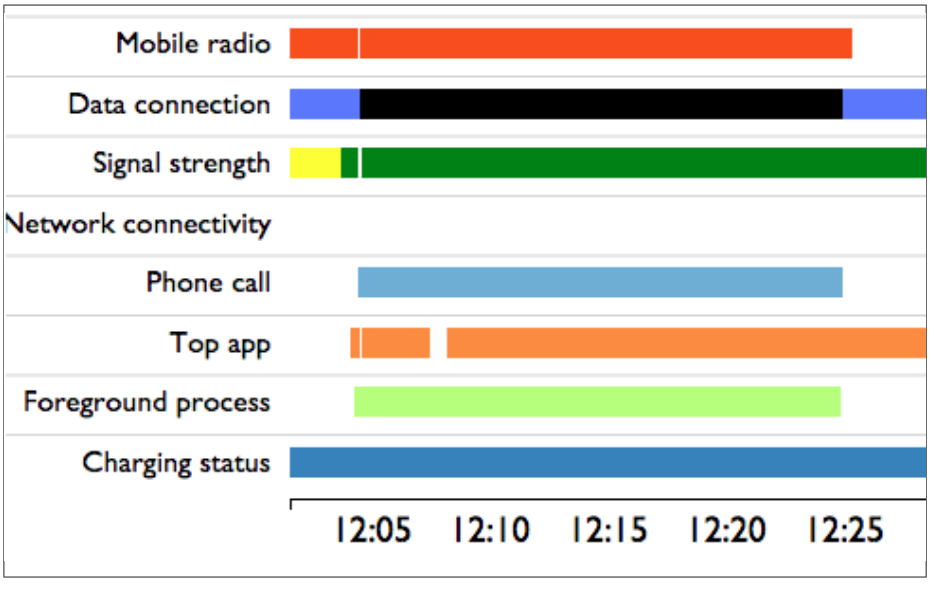


Figure 7-2. Battery Historian showing Circuit Switched Fallback

Testing Tools

So far, we have discussed the power usage of Android's radios, and how mobile data networks work. How do we use this knowledge to optimize our Android application 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) also records packet captures, but its primary focus is to help cellular application traffic and performance, and thus a valuable tool.

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 realtime, 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.)

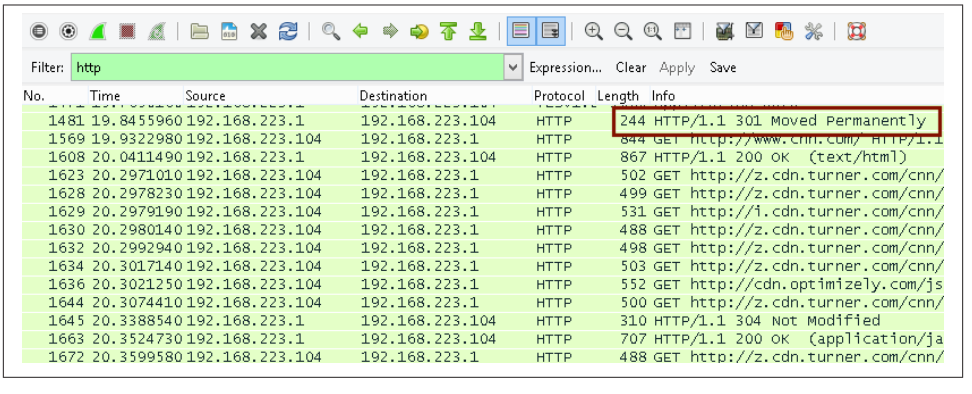


Figure 7-3. Wireshark Packet Capture

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 3 such redirects.

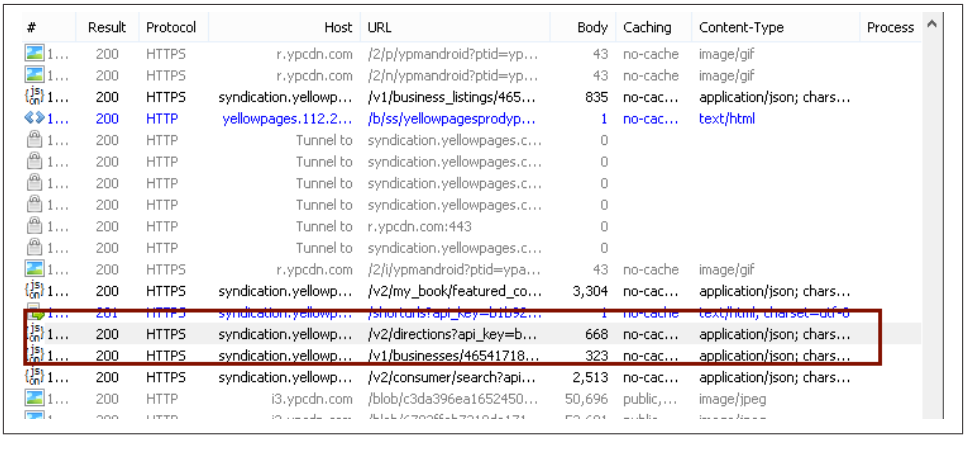
The filtering tools in Wireshark are extremely powerful. It is possible to search and filter for specific files, all PNG files, files with cache headers, etc. (the possibilities are endless!) In practice, these are all great searches, but you have to have an inclination to what the issue is - otherwise I kind of feel like I am looking for a needle in a haystack with this approach. However, if you think you know what your issue is, Wireshark is a great way to drill down and pinpoint exactly the issue.

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 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 certs 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 below, 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…	200	HTTPS	syndication.yellowp…	/shortcuts?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, since 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!) Knowing that this store is about 10 minutes away, the driving time appears to be reported in seconds (661s ~ 11 minutes.)

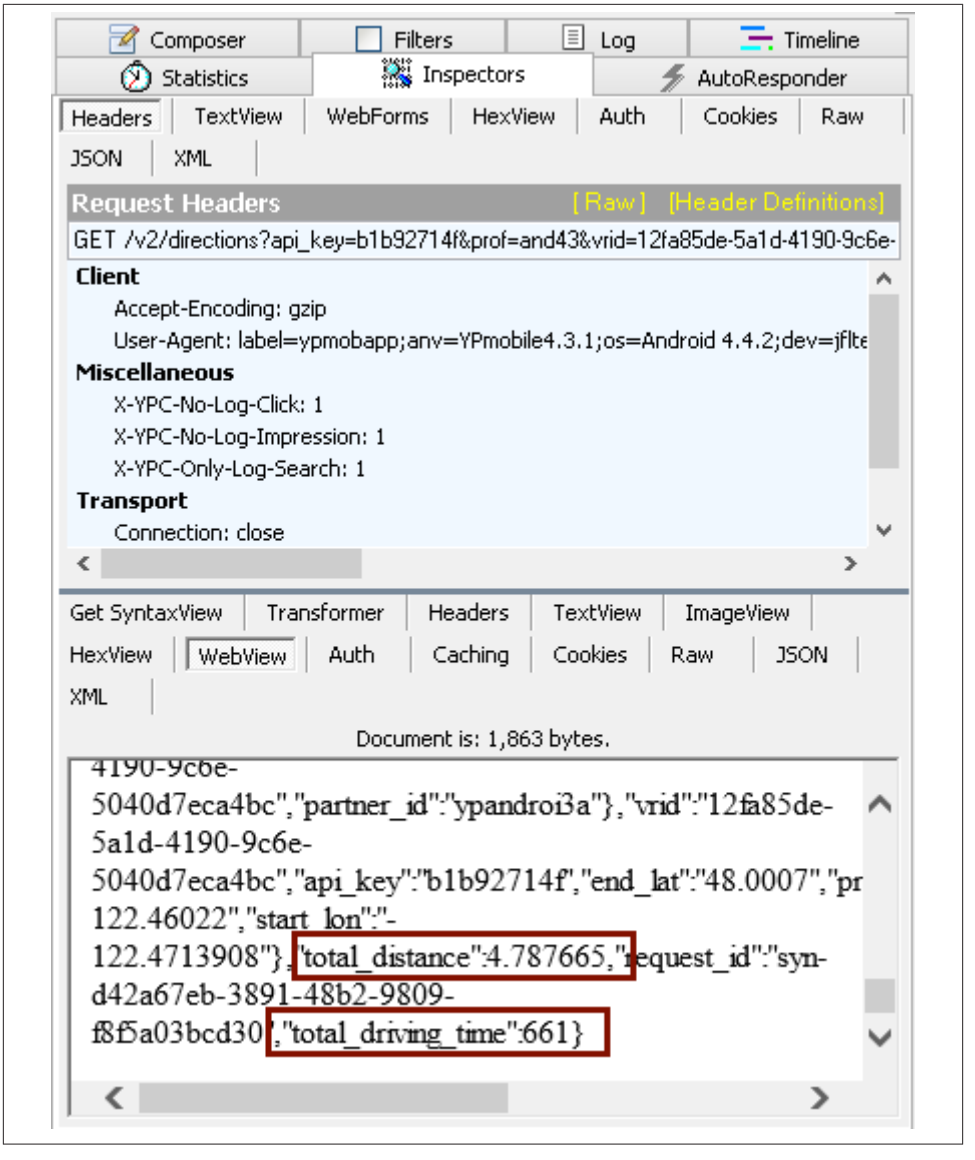


Figure 7-5. Fiddler Proxy Capture detail windows

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 customer's data. By decrypting the data, Fiddler and

MITMproxy also allow you to ensure that the correct files and information are being sent to any 3rd party SDKs that you have added to your application.

AT&T Application Resource Optimizer

The Application Resource Optimizer (ARO) is a tool specifically designed for monitoring network performance in Android and iOS applications. It is a free/open-source tool from AT&T, and it contains much of the same packet capture functionality of Wireshark and Fiddler. Additionally, ARO has the ability to collect packet traces over the cellular network (while Wireshark and Fiddler require Wi-Fi connections to a computer.) 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 on performance improvement. The summary of these tests is shown in Figure 7-6 where 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.

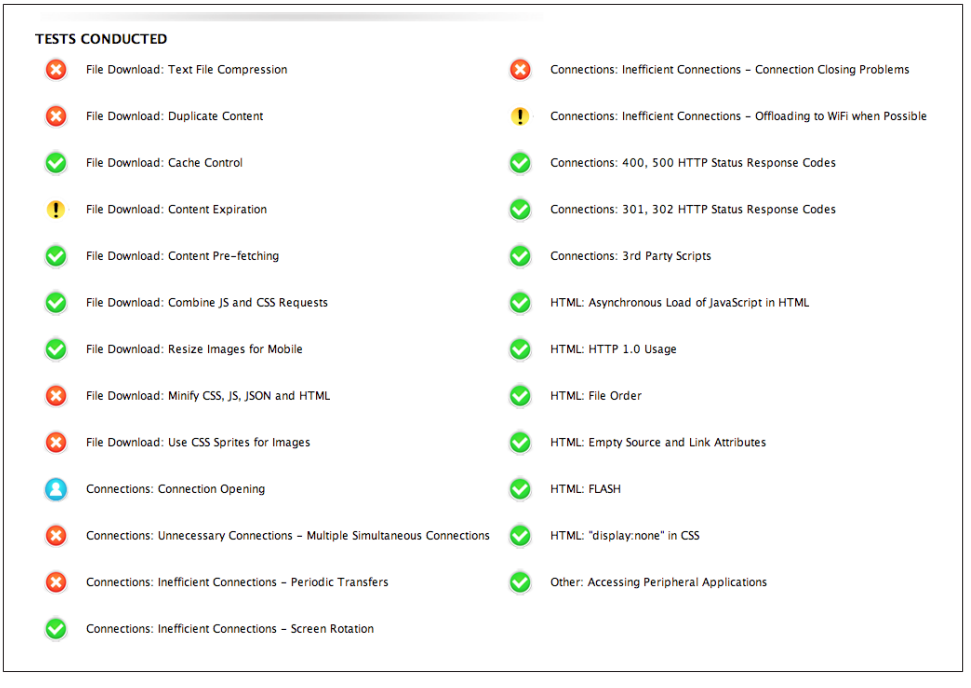


Figure 7-6. ARO Best Practices: Pass Fail

There are 2 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 application (if there are a lot of applications 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 application 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:

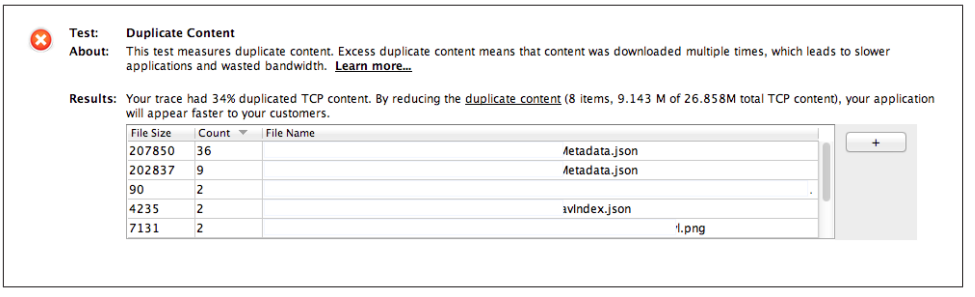


Figure 7-7. ARO Duplicate Content Best Practice. 34% (9MB) of data sent multiple times is a lot!

There are five tabs of data provided out of each trace, but the Diagnostic tab is where oyou can really see the data the flows in and out of your application.

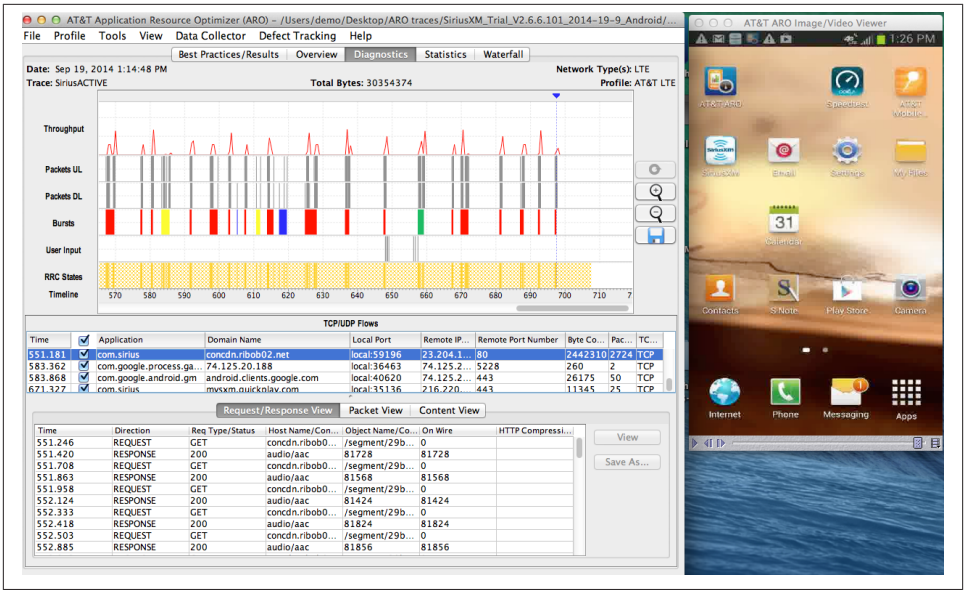


Figure 7-8. ARO Diagnostic Tab

Inside Figure 7-8 there is 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 race, 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 diagnostic 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 Figure 7-1. 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 wold see the line move to the right - allowing you to see the packets being transferred while also watching what was on the device screen.

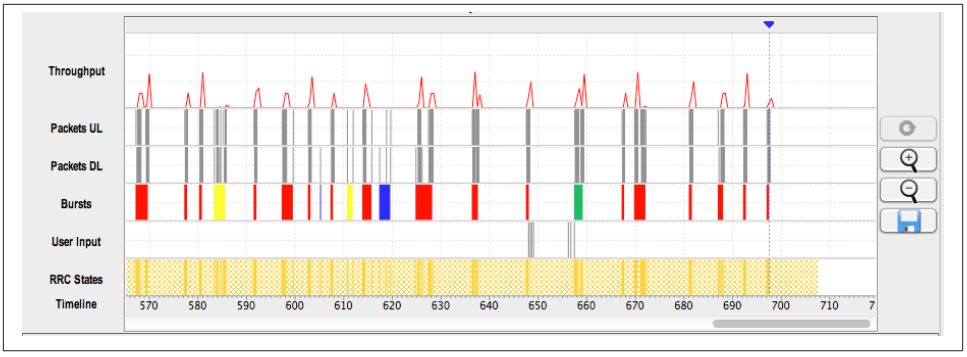


Figure 7-9. ARO Diagnostic Tab Graph

The two tables below the graph provide more insights into each data connection that occurs in the network trace. The top graph shows every TCP or UDP connection that was initiated during the trace. Currently highlighted is a connection started at 551s from the SiriusXM Radio application. 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 81KB music files transferred in this connection.

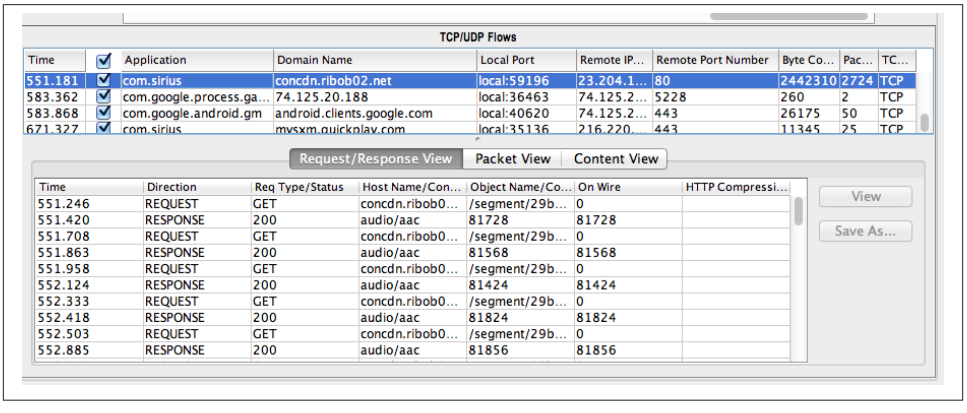


Figure 7-10. ARO Diagnostic Tab Tables

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 application 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 application experience. This native app generally just displays the content in an embedded webview to appear as if it were native. Since the wrapper is not possible to optimie, 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

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 particular order of importance, as each of these will affect application performance differently (and some will likely not apply to your application 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 application performance basically derive themselves from Steve Souders' iconic 2004 list of 14 performance rules from his book High Performance Web Sites (O'Reilly, 2004):

- 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 optimiations, and we will cover them in the next sections.

File Optimizations

There are 2 basic ways to download data faster: lower the number of requests (Souders rule #1), and/or reduce the size of those requests (several of the Souders' rules). This can be a hard pill to swallow, since 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 application.

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 application, 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 round trips and faster delivery of the file. For example, in one application, we saw a 200KB text file downloaded without Gzip compression. Placing this on a server with compression enabled reduced the size on the wire to 51KB. 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 applications. 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 pre-compressed files" warning.)

Enabling Gzip compression is a simple server change (no code change in your application.) By simply adding the file extensions/Mime types etc. 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:

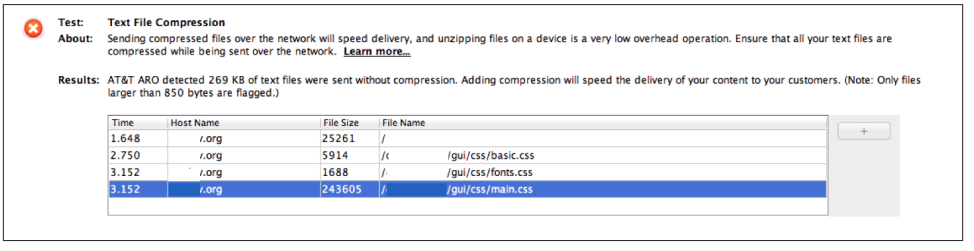


Figure 7-11. ARO Text Compression Best Practice

1. Best Practice: Text File Compression. Any text file sent without compression is listed here. 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 (since they fit into one packet, and will only require one round trip - even without compression).
2. Diagnostic 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 filesize by 10-15%, but the difference in Gzip savings for the minified vs. not-minified file might only be 1-2% (since white-space compresses well).

However, you should always minify before Gzipping, not to save the 1-2% of network transmission, but consider the storage savings on your customer's device. In addition, reading a smaller file into memory is faster and (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, JS, JSON and HTML files for minification opportunities. It calculates the potential savings on each file, and provides a total savings for the files captured in the trace.

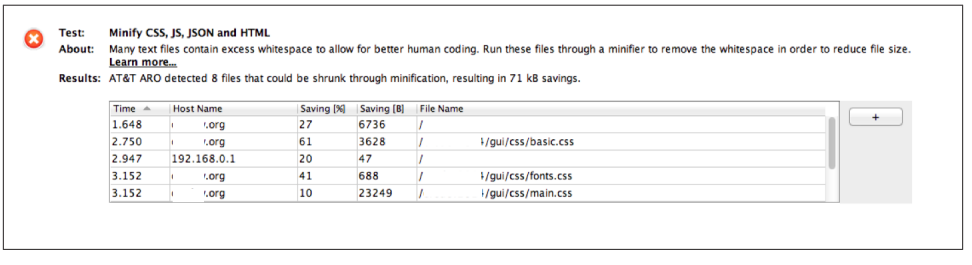


Figure 7-12. ARO Minification Best Practice

Images

When it comes to the apps, images are the most common file downloaded. They are also amongst the largest files in size, and since images are everywhere, easily recycled from your webpage or other digital service. Controlling image size is a fantastic way to reduce the data usage in your mobile application. There is a balance between image quality and image size that you must discover for your application (probably with the help of UX and editorial teams.) Once you find that correct balance for your application, you'll have a great looking application, where the images are optimized to download and render quickly.

Super Size It?

If you create one version of every image in your application, 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 application 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 250kb images for the thumbnails next to articles. With 6-8 article titles on the page, these tiny images added 1.5-2MB of data on every startup of the application. 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 application 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 describing 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 application 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 applications focused on photography and graphics, you may decide to not do any lossy compression. The amount of compression is a delicate balance of editorial/UX wanting as clear as possible, 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 your application, 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 application. 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 2 dimensions to caching: First you must turn on caching in your application 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. The Cache-Control header has a few common values hat you can assign:

1. 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.
2. no-store: If your files use this term, the files cannot be cached, and thus must be downloaded every time.
3. 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.
4. 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, 3600 (1 hour), 86400 (1 day), 3153600 (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 86,400s (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 86,400s.

```
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 example above, 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 application 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 Figure 7-7 shows a list of files downloaded more than once in a trace, the # 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 up 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 application'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 applications whose cache is not configured (or configured correctly) on the device.

If your application is downloading content in a duplicate manner, take a close look to ensure that the headers are populated properly (server fix), and that your application 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 167 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 application. 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 application? 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 application 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 application 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 3rd 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

Since 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 application (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 application to reduce the number of connections that the Android application made in the background. In this early version of the app, we saw 3 connections running in an uncoordinated way in the background. Every 30 minutes, these 3 connections turned on the radio 7 times. In the figure below, 30 minutes are bordered by the packets with red "Bursts." In the top example, you can see two closely (but not overlapping) purple, yellow and blue bursts. The purple connections open the 2nd and 3rd connection, the yellow bursts are reusing these 2 connections, and then then blue is a packet form 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 application. The bottom trace shows the improvement. The "updated" version of the app pushed all 3 connections into one transaction time, and the developers doubled the refresh rate to every 15 minutes. Now, with 2 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 application installed!

Not every developer has the time to build their own transaction managers, but the Android developers have been listening. In (to come), we looked at the "JobScheduler" on page 62 API 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:

```
public static String getNetworkClass(Context context) {
    ConnectivityManager cm = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_S
    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 radio was already in use, I used the following:

```
 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 ConectivityManager.isDefaultNetworkActive(). Using a radio connection that is already established is a great way to share the resources, and save customer battery.

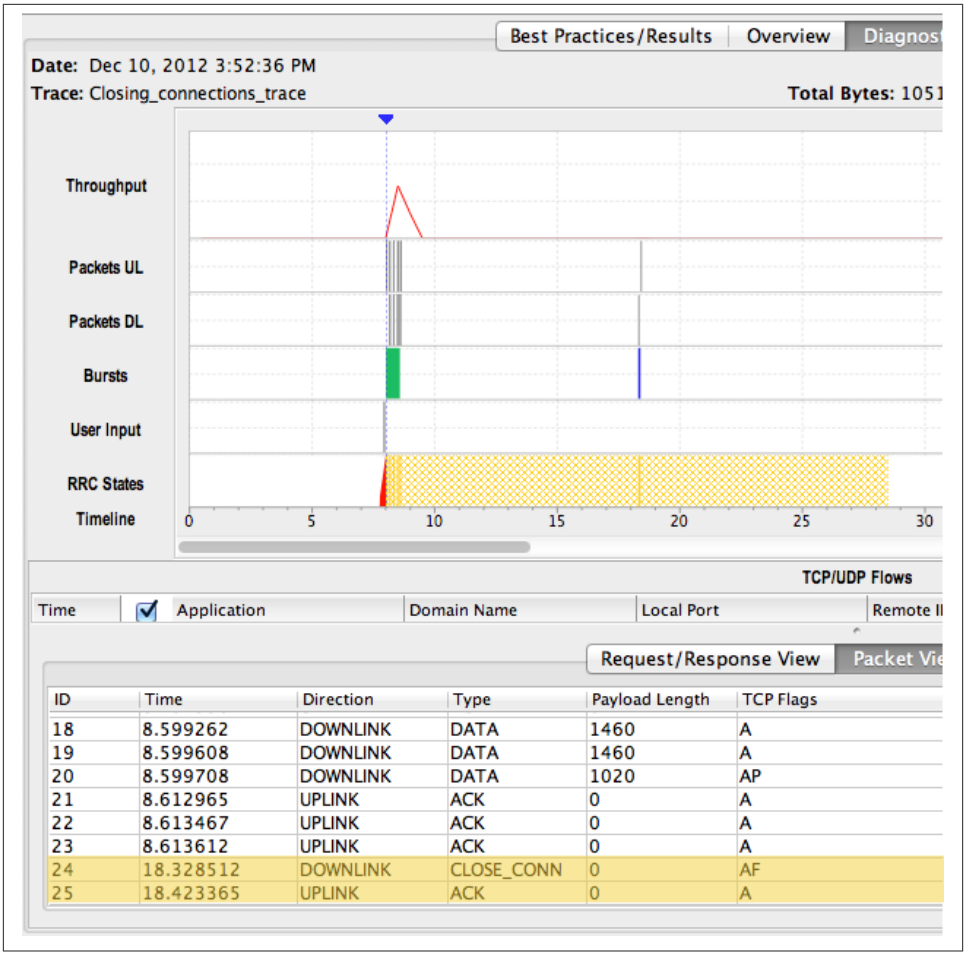
GCM Network Manager

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 "JobScheduler" on page 62's APIs 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 15s 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 10-15s RRC State Machine on the device, causing extra battery drain for your customers.



ID	Time	Direction	Type	Payload Length	TCP Flags
18	8.599262	DOWNLINK	DATA	1460	A
19	8.599608	DOWNLINK	DATA	1460	A
20	8.599708	DOWNLINK	DATA	1020	AP
21	8.612965	UPLINK	ACK	0	A
22	8.613467	UPLINK	ACK	0	A
23	8.613612	UPLINK	ACK	0	A
24	18.328512	DOWNLINK	CLOSE_CONN	0	AF
25	18.423365	UPLINK	ACK	0	A

Figure 7-14. ARO Diagnostic Tab showing Connection Closing Issue

In the above screenshot from ARO, a small image is downloaded at 8s, but the connection is not closed. At 18s, 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 18s comes from the server to close the connection). Instead of turning off at ~18-19s, the radio remains on until 28s - 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. The the code snippet below, 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.).

```
HttpURLConnection connectionCloseProperly = (HttpURLConnection) ulrn.openConnection();
connectionCloseProperly.setRequestProperty("connection", "close");  //this disables "keep-alive"
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 close the connection.

Regular Repeated Pings

For applications that require data updates at regular intervals, tools like Google Cloud Messenger should be used to push this information down to the application. 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 application (and the device) asleep.

There are times (think real time games) where the application 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 application is running can cause major battery drain.

A Perfect Storm: Repeated Connections and Closing Connections

Imagine an application that sends realtime data every 5 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 90s 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 5 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

application 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 vs. HTTPS)

When transporting data over the network, it is imperative that you keep your customer's private data secure. It seems that not a week goes by without a serious breach of private information from a mobile application. 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 send 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 addition round trip 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 described above, 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 customer's network. We (obviously) cannot control how or where our customers connect to our application. 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).

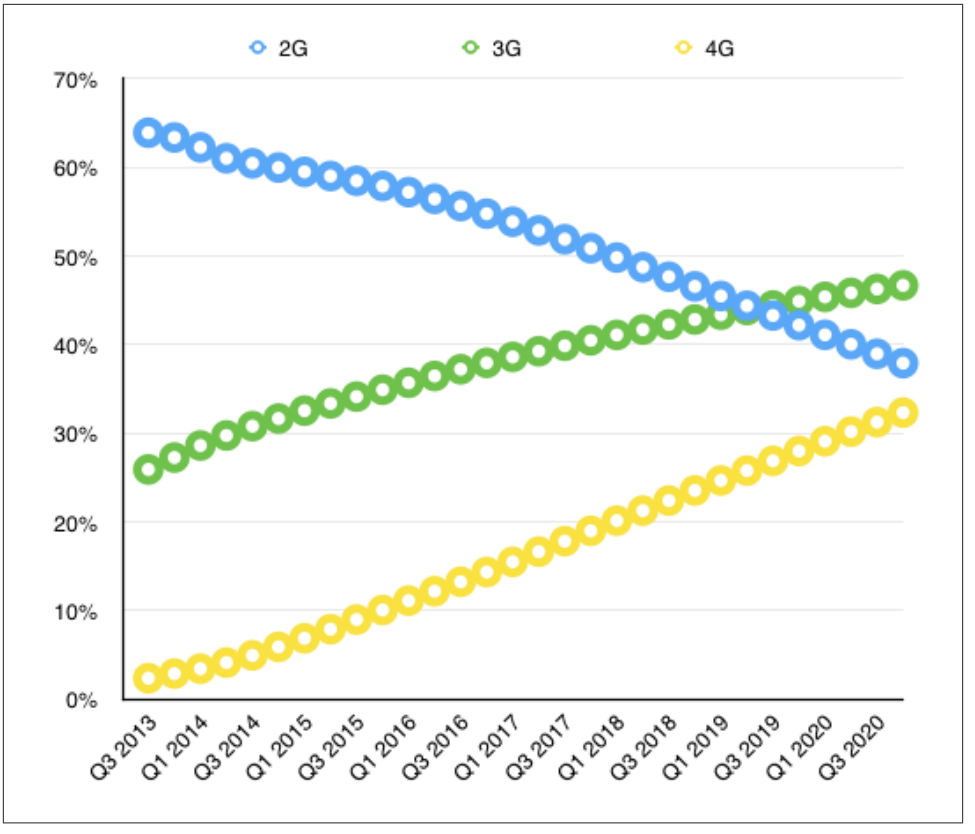


Figure 7-15. Global Market Penetration by "G" (Courtesy GSMA Intellligence)

In examining the 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 on are running exclusively on 2G networks. 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 on exclusively on 2G for data connectivity.

Smartphones running on slow networks is not a problem to 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 application, so it makes sense to ensure that your mobile application runs well on slower, more congested networks.

In Chapter 2, I discussed how "Your Device is Not Your Customer's Device" on page 13. 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 application. While the speed of light is incredibly fast, it still takes 53ms to make a round trip from Boston to London (and Boston to Sydney is 162ms!) In order to reduce this latency, a 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 Round Trip time to request and deliver the files.

Let's look at that Facebook example above. 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 round trip time in a fiber cable takes about 32ms (assuming 200,000km/s speed of light in fiber) from the local CDN. 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 176ms (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 Application 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 application is used. (Hey, it doesn't hurt, right?) Facebook has published reports of testing done in Africa and Indonesia and some of the learnings that came from them. In Africa, they found that their application burned through their 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!)

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. They dis-

covered that they needed to be aggressive on CDN mapping. Looking at the slowest connections, they found that only 16% of traffic was coming from local CDNs, and a full 84% of images/videos were coming from CDNs literally halfway around the world (adding significant latency). By re-mapping the traffic patterns, they were able to significantly improve the speed of content delivery.

Since few companies have the resources of Facebook, 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 used 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 plugin 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. The Network Attenuator runs on a Samsung S3 ICS kernel (requires root and flashing of a custom ROM, provided by AT&T). Once installed, the application 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 Round-trip time for each connection.

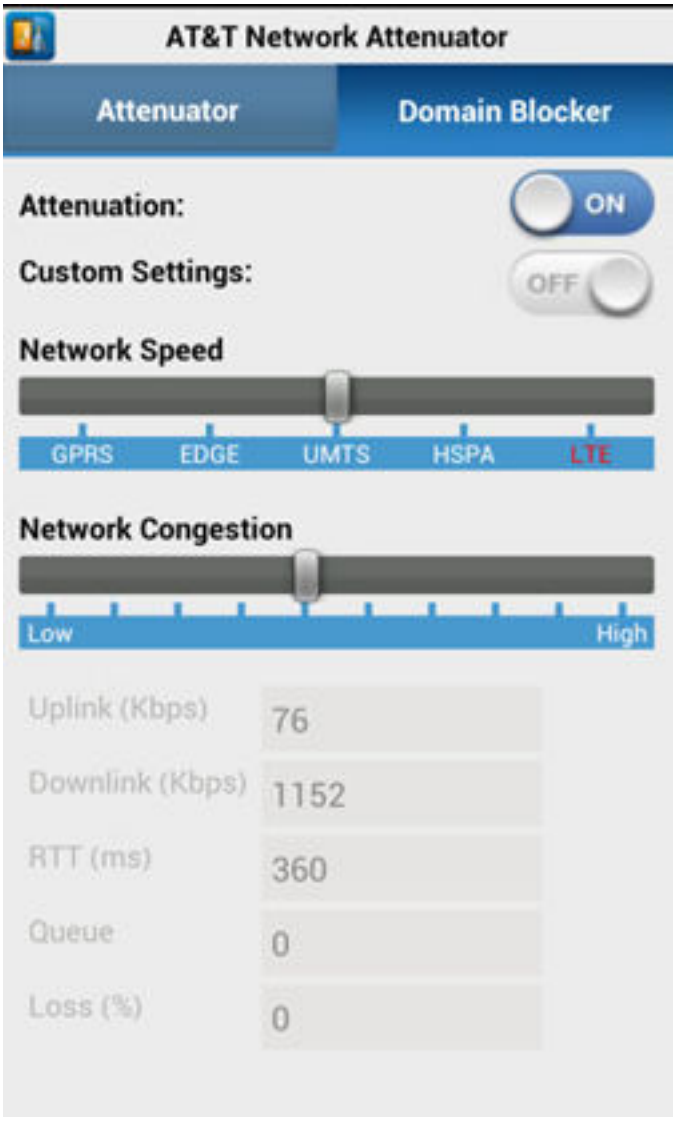


Figure 7-16. Network Attenuator APK

Building Network Aware Applications

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 application on 3G or 4G, but there are tricks to enhance the experience on slower networks. I

like to call application that utilize this architecture "Flexibly Network Aware" (FNA) applications because the apps 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 application.)

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, reducing the number of images (or at least varying the image size.) By querying the TelephonyManager, you can use your application to vary the type of experience to display:

```
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 application 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 "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, since your customers are not hindered by a cost for each MB of data that you send over Wi-Fi.

```
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 2 times: the time until a 200 OK response form the server, and the total time it takes to download the image. Responsetime (the time to getting a 200 response from the server) is equal to 2 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.

```
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();
           Long gotresponse = System.currentTimeMillis();     //time 200 response received
            }
           final HttpEntity entity = response.getEntity();
                       contentlength = entity.getContentLength();//get ContentLength of file
           if (entity != null) {
              InputStream inputStream = null;
              try {
                                       inputStream = entity.getContent();
                                       final Bitmap bitmap = BitmapFactory.decodeStream(inputStre
                       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)imag
                       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 3 different networks:

Table 7-2. Download Times (s) of Images

File	LTE	UMTS	EDGE
Big (143 KB)	1.938	5.243	9.405
Med. (41 KB)		2.793	
Small (27kb)			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 application to flex with the network in near realtime, 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 application 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 (for example, of there are only 2 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 Application On Slow Networks" on page 197, 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 pwoer 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 applications, a precise location is needed, and the GPS radio will turn on to recieve 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 log file 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 the /sdcard/btsnoop_hci.log file.

Opening this log file 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:

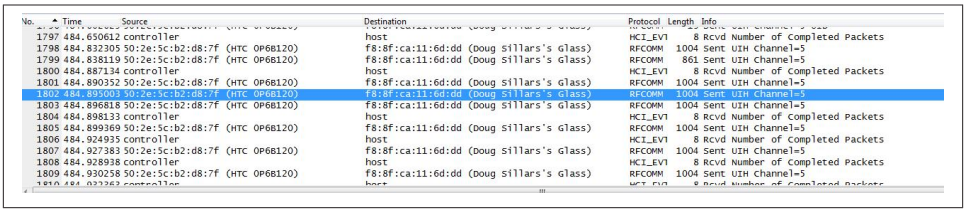


Figure 7-17. Bluetooth TRaffic in Wireshark

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":

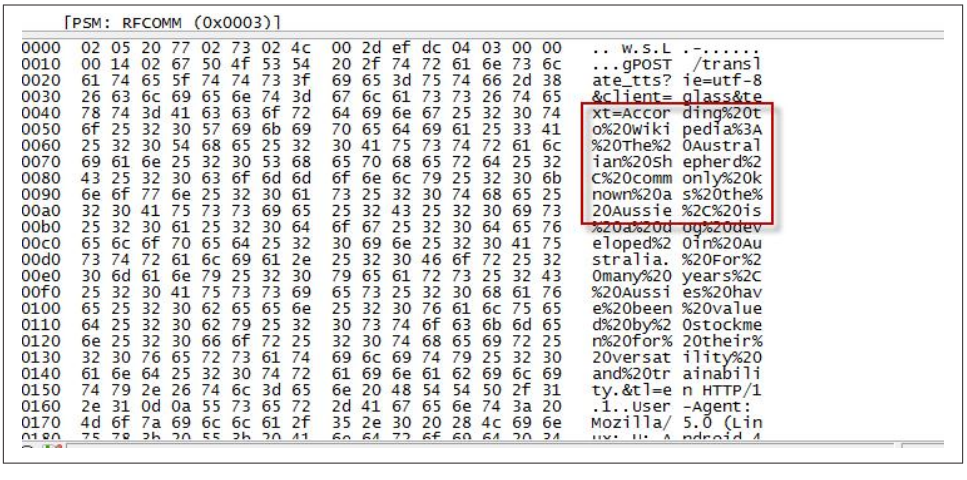


Figure 7-18. Bluetooth POST response

Using Fun Wireshark tricks you can quantify the packet and data transferred over time over Bluetooth:

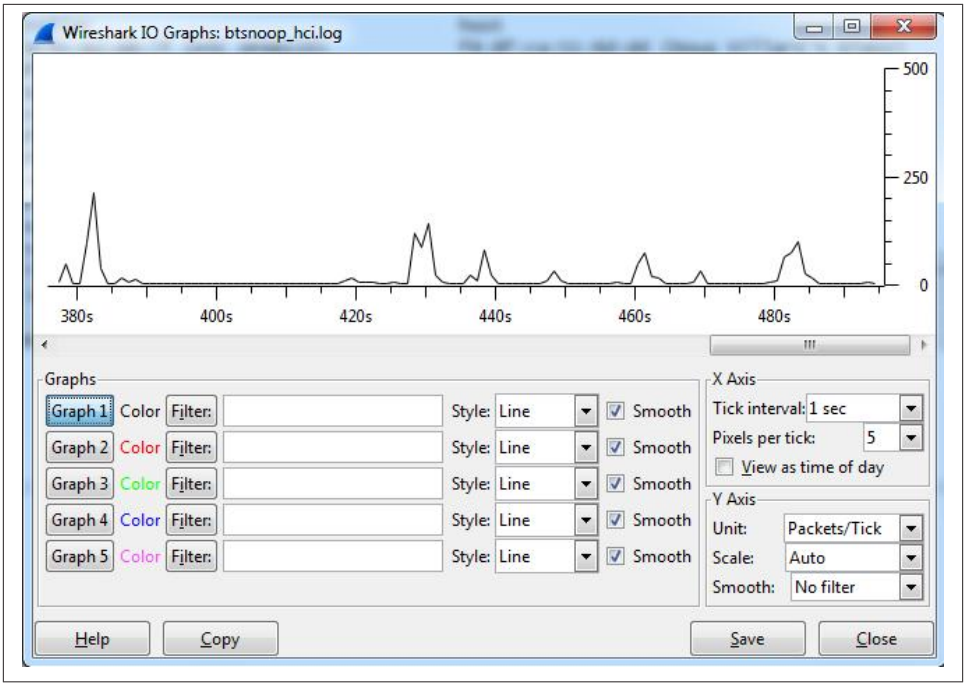


Figure 7-19. Bluetooth POST response

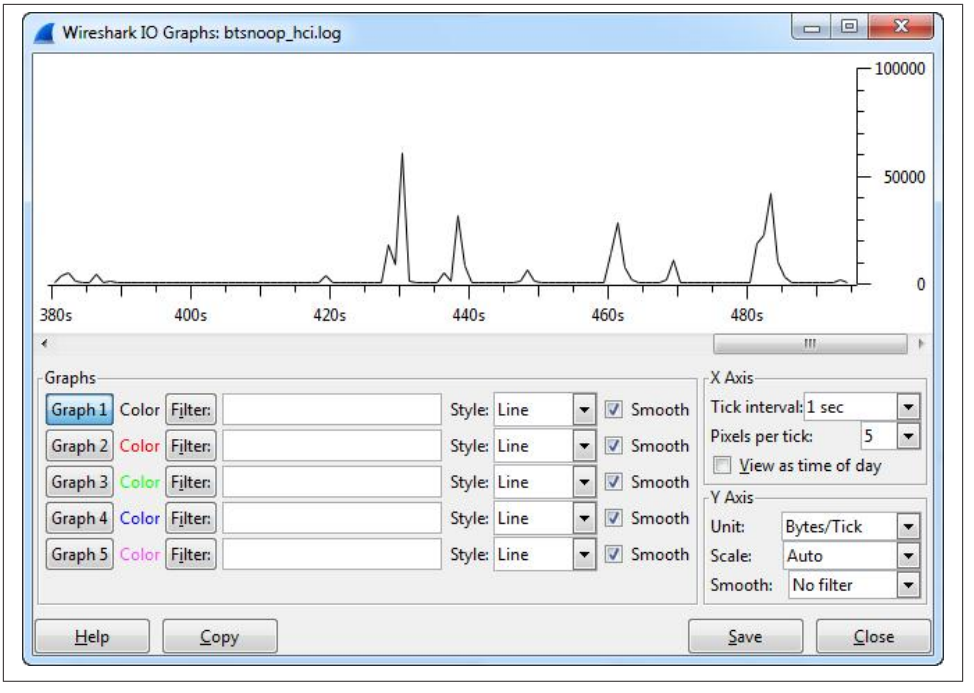


Figure 7-20. Bluetooth POST response

Conclusion

Most Android applications use the cellular and Wi-Fi radios to communicate to outside servers for information. Since the radios are second to the screen for battery drain, it is important to use them judiciously. Further, since most users have a monthly cellular data allowance, it 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. By ensuring that your data transfers are optimized for the available network will allow your application 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 customer's location will save customer battery life and enhance the user experience around the world.

CHAPTER 8
Real User Measurements

In the previous chapters, we have walked through a number of great tools that can be used to diagnose issues in your Android application. We've looked to optimize battery, memory, CPU, 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 we insure that your application is performing optimally for all of your customers - regardless of location, network or device? The answer is to collect runtime data on your application, aggregate the results, build reports, and look for issues that might arise from the data. These analytics are drawn from the application itself, and is commonly known as Real User Measurements (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 application 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 application 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. Since you are reading the book, 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 application, and will provide you with insights to help you improve the usability and performance of your application. 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 in the market available. 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 application. 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 3 RUM SDKs to add to a sample app (ImageScroll) to see what data I can obtain.

The first SDk to be added is Crashalytics. It uses a very simple widget, pictured below. All you must do is click the install button, and the code will be automatically added to support these analytics to your app:

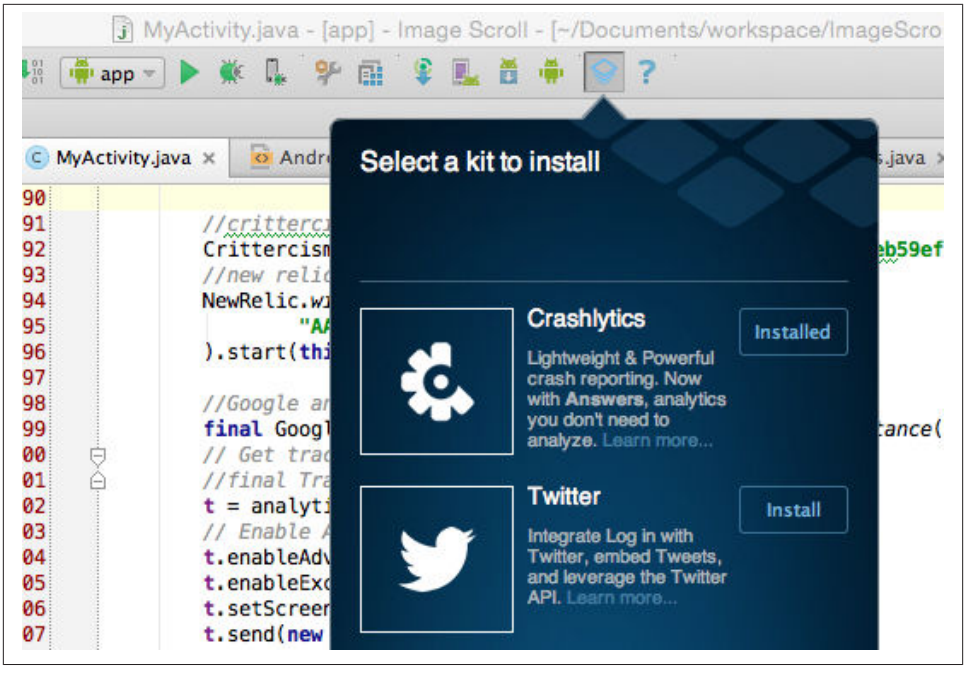


Figure 8-1. Installing Crashalytics

Once the SDK is installed, you simply build and distribute your application. 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 application 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 and app versions, etc. To get sample RUM data, I built an application called "Image Scroll" that simply loads 10 images at a time as you scroll through them. 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 application crashes. This is by design, in order to track app crashes across devices.

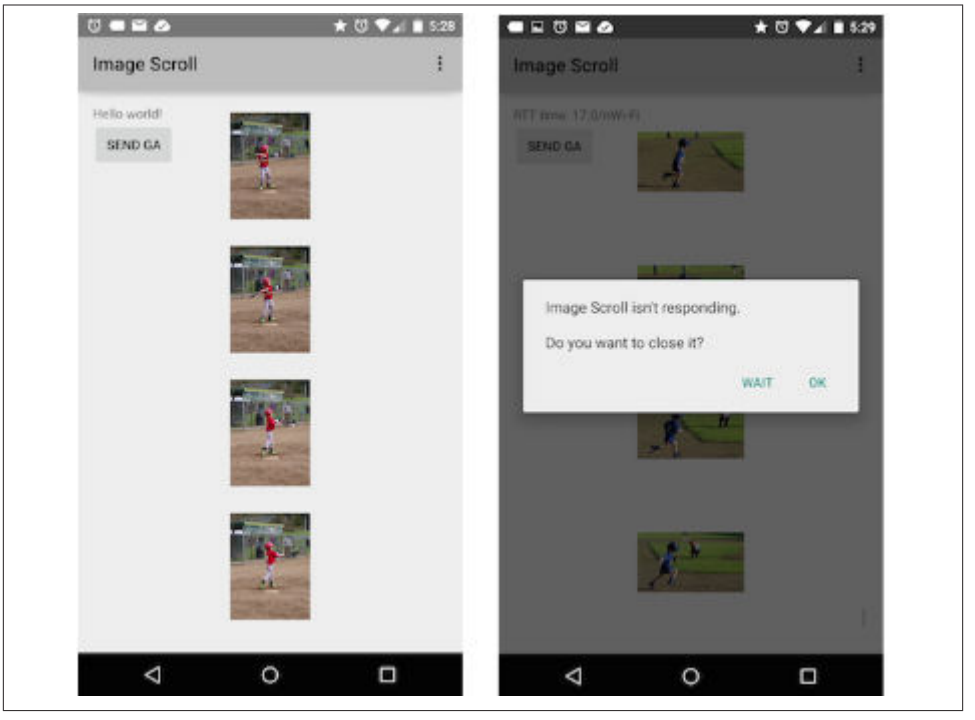


Figure 8-2. Image Scroll App

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 2nd group (900 KB vs. ~50KB) as test cases for errors that affect performance.

This sample app has (in alphabetical order) Crashalytics, 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 application 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 optimised for data usage and battery drain, as we'll see in "RUM SDK Performance" on page 225.

Crashing

As discussed in Chapter 2, performance is crucial to customer satisfaction of your application. Actually seeing 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 log files 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 lets force it to crash
           // if(totalImageCount ==100){
           //     totalImageCount=0;
           // }
                   //if totalImageCount reaches 100 the app crashes since I have exceeded the arra

           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 round trip 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 Array Index (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: 550346640083-0001-35CC-27DD9B9DA026.cl
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 application 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 Crashalytics and New Relic. I have noticed (using network monitoring) that Crittercism reports generally occur a short time after the application 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. Since 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 Crashalytics.

Examining A Crashalytics Crash Report

When Crashalytics finds a new crash, you immediately get an e-mail (hint: build a special *crash report* e-mail address or filter.) reporting a new issue was found. Clicking the link in the e-mail takes you to the crash details webpage. here is a screenshot of the Image-looper crash:

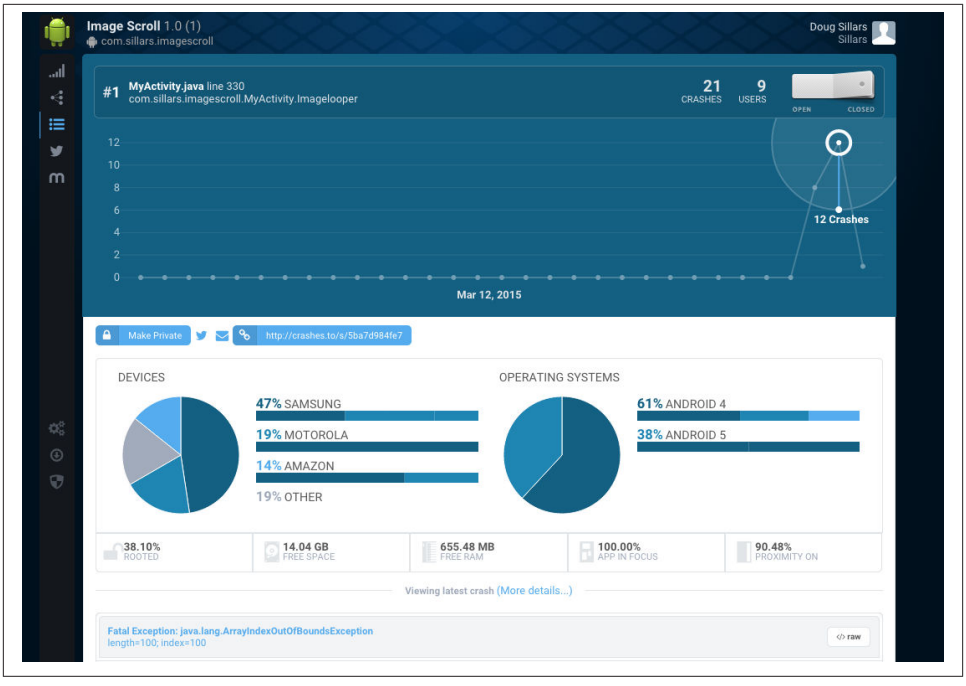


Figure 8-3. Crash details

The top of this dashboard shows the number of crashes and the number of users (in this case 21 crashes across 9 users), with a graph showing the count of crashes over the last 3 days (highlighted are the 12 crashes form March 12, 2015). The pie and bart 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), you can break down devices further (and the same with the OS versions):

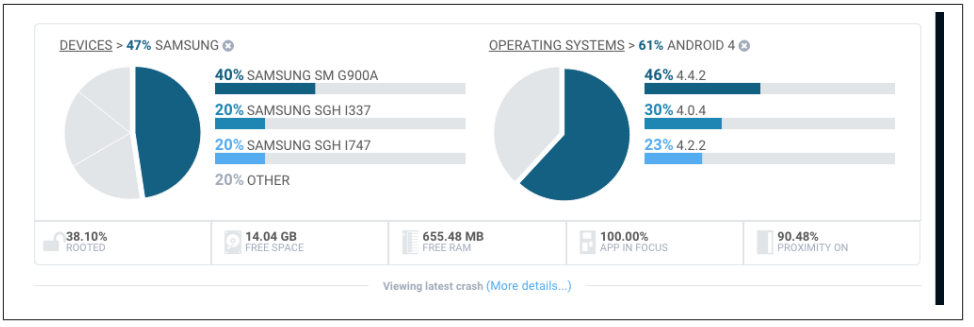


Figure 8-4. Device Breakdown of Crashes: Focusing on Samsung devices and Android 4 OS versions

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 application is performing around the world. Tools that report how your application performs on different devices and networks around the world can help you isolate issues, and find bottlenecks in your application that your traditional testing might not find. Perhaps your application 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 application that you can capitalize on for future releases. There are a number of SDKs and tools tha report this information. Below is a report of app usage in the last 24 hours from Crittercism:

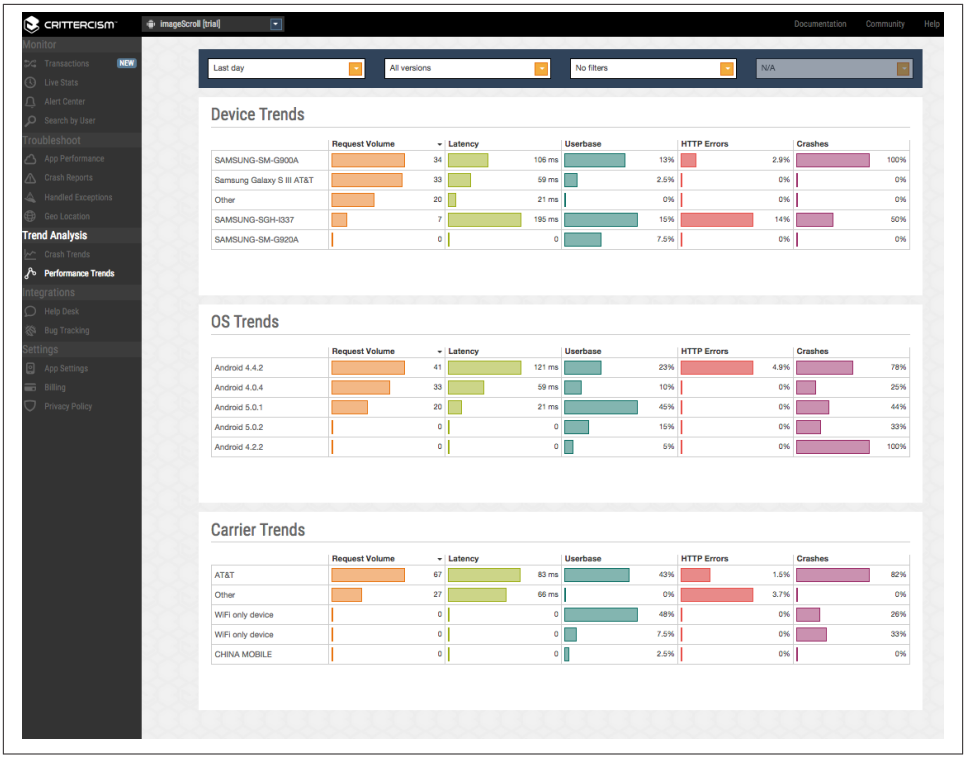


Figure 8-5. Crittercism Report: 24 hour Usage Trend

This dashboard breaks down the # or 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 application 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 since 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 20s (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 overlaoded, or that the files migh 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 5 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 (are your servers doing ok?). Filters for carrier allow you to see if certain regions might not be getting the data fast enough (recall in "Testing Your Application On Slow Networks" on page 197 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 200ms, but deeper investigation shows that the slower connections are analytics, while my image provider (Photobucket) has a response time of 23ms. The error rate is due to the fact that 2 of the 100 images loaded by my application have incorrect urls, throwing a 404 error. Seeing any error here should prompt you to dig deeper.

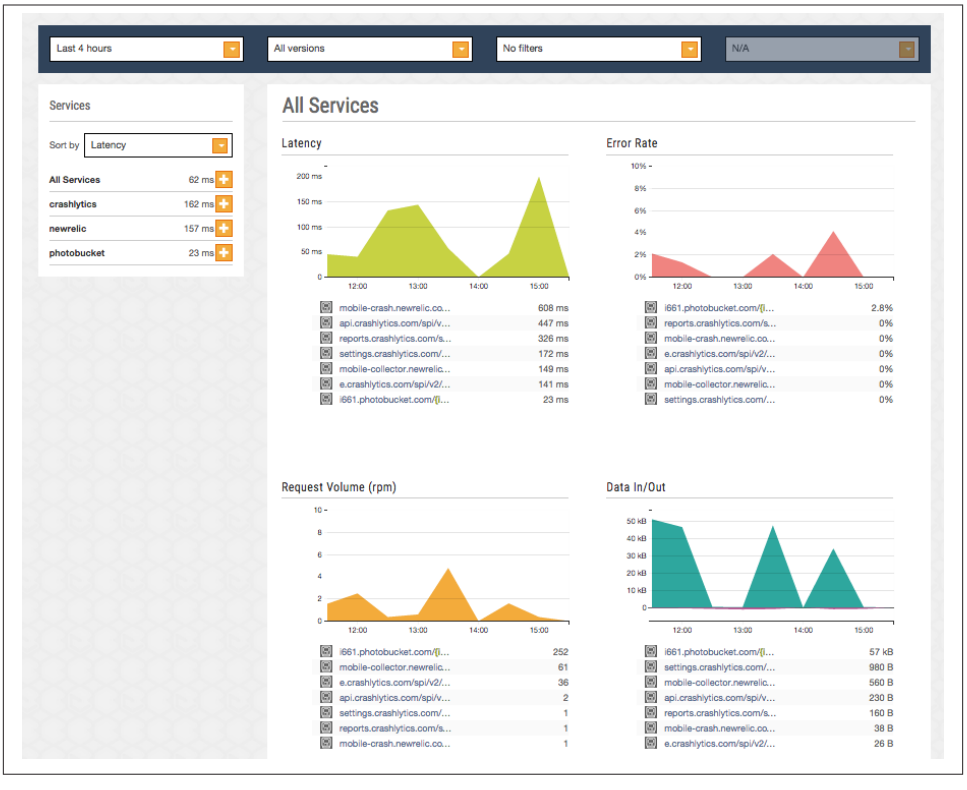


Figure 8-7. Latency, Errors, Volume and Data Graphs

The New Relic tools have a list of all network errors. In the figure below, 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.

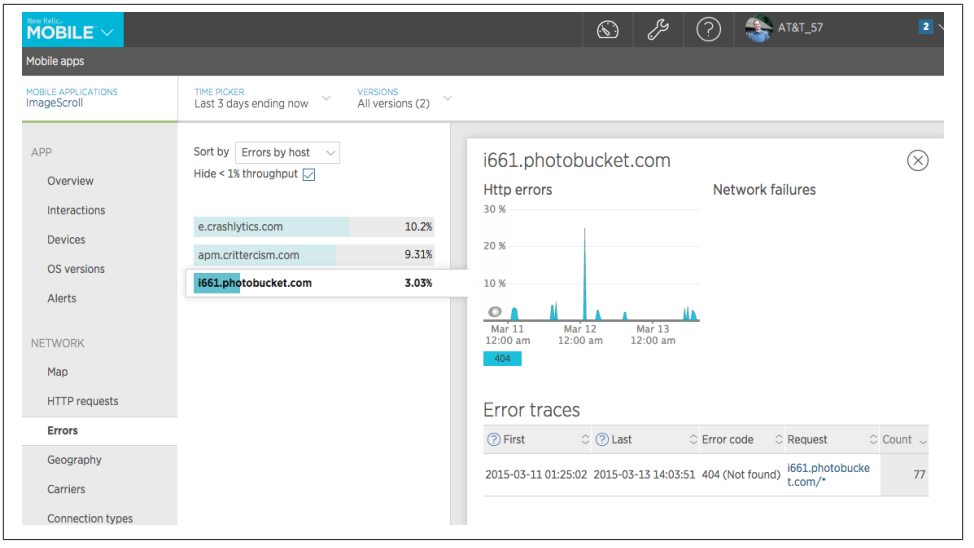


Figure 8-8. Network Errors

Being able to correlate HTTP errors from your application with your server logs allows for very powerful troubleshooting. You can narrow down the issue to a platform, or version of your application 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 application (and how long), how often they use it and more. By better understanding who your users are, and how engaged they are, you have more opportunities to improve your app.

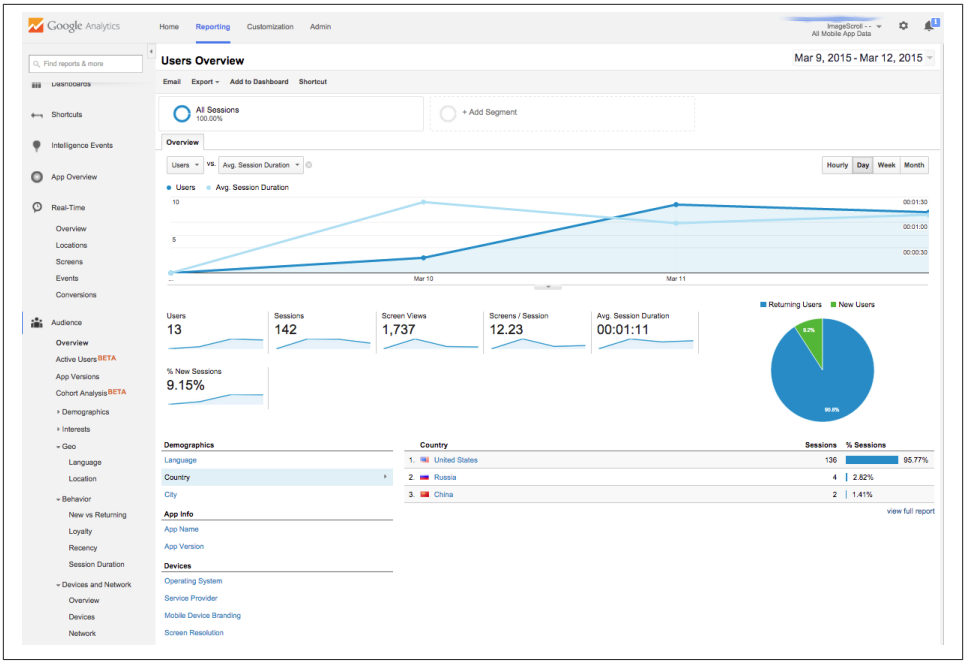
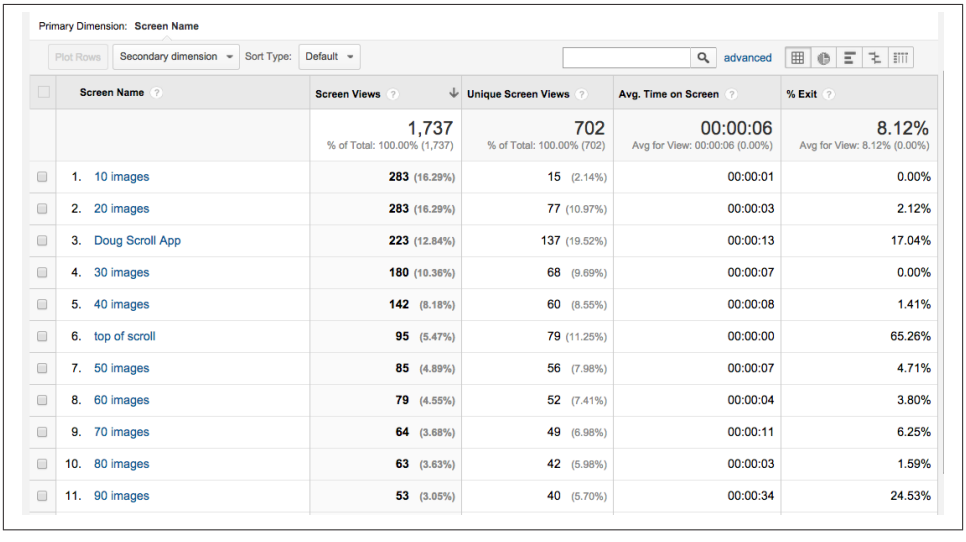


Figure 8-9. Usage Informaation

In the above image, Google is giving us a lot of information about our users. The top graph is showing the daily user count vs. the average session length (on March 11, there were 9 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. Form tis information, 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");
```

Now every 10 images, Google adds a screen view and Crittercism adds a breadcrumb for issue tracking. Let's look at the dashboard report form Google Analytics. 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 application, 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 application - 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 application 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.

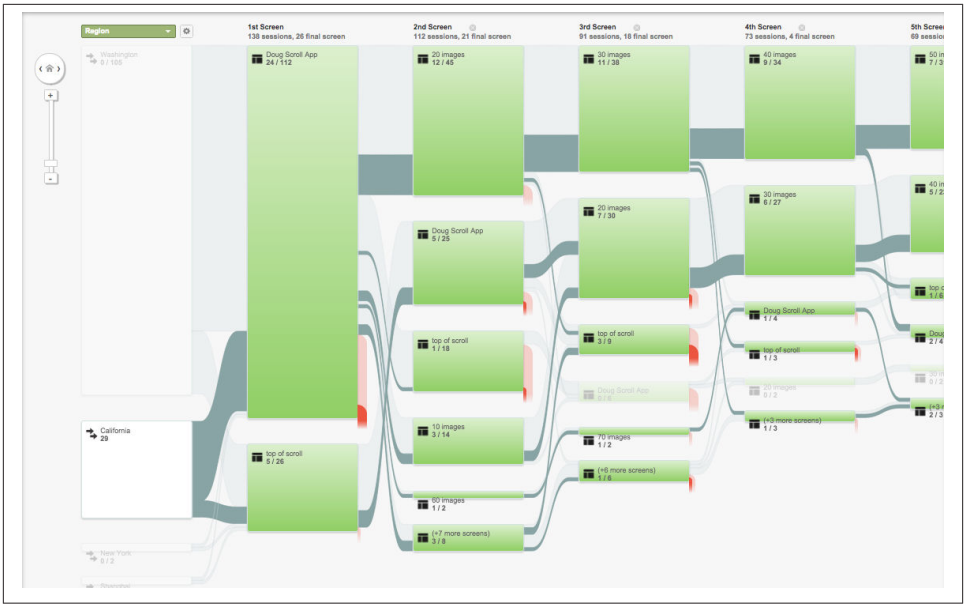
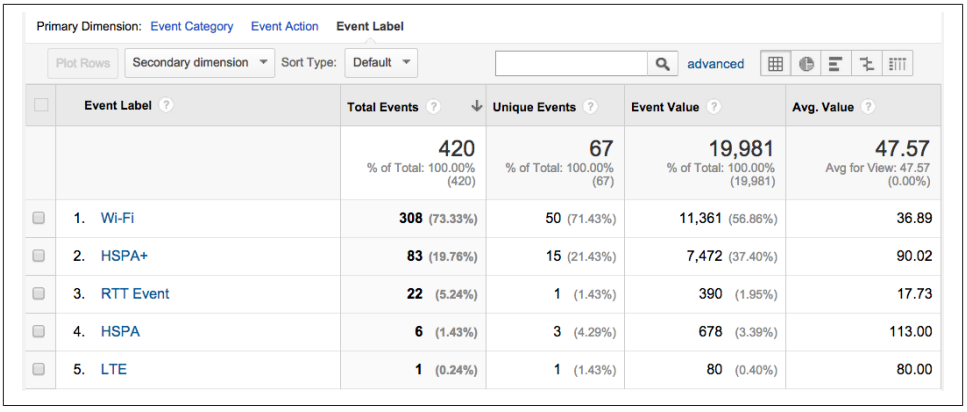


Figure 8-11. Behavior Flows (California Users Highlighted)

You can find other pain points by setting unique events that are reported back to the analytics server. In Chapter 7 we used ??? to identify the Network type used by the customer and ??? 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 the following:



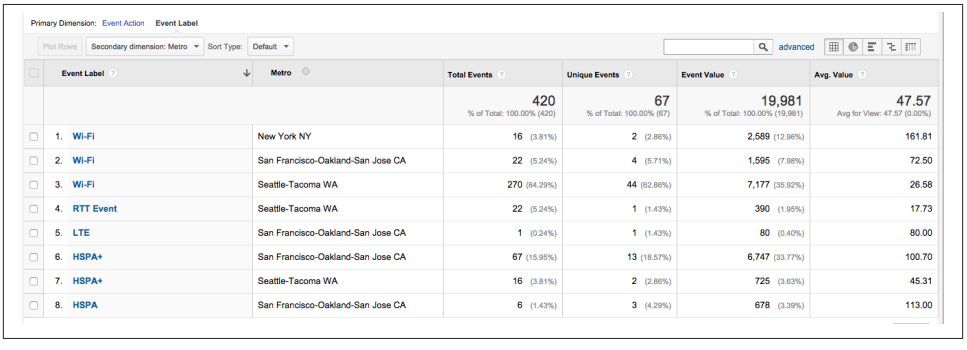
Primary Dimension: Event Category Event Action Event Label

Plot Rows | Secondary dimension | Sort Type: Default | advanced

	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 round trip time is reported in the last column of Figure 8-12. The average Round trip time might not be extremely useful, since 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. Here is the data by US Metro region, showing that the Wi-Fi in Seattle has a faster RTT than that in SF and New York:



Primary Dimension: Event Action Event Label

Plot Rows | Secondary dimension: Metro | Sort Type: Default | advanced

	Event Label	Metro	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	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 are being sent regularly, you can track users in near real time. All of the providers discussed above show the performance of apps in near realtime. In the example below, we see 4 app loads (blue line), and one crash (red line) in the last 30 minutes.

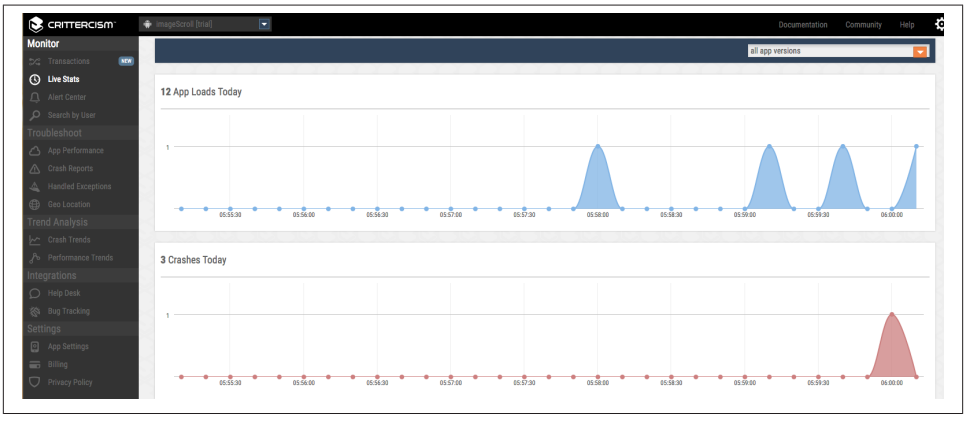


Figure 8-14. Real Time Analytics

Big Data to the Rescue?

As your application 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 hep 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 measurements 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 application, and report how performance improvements actually improve the retention and time spent in the application.

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 round

trip times are important for user critical data, but longer round trips 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.

Running a network trace to look specifically at the SDK traffic using the Application Resource Optimizer "AT&T Application Resource Optimizer" on page 176 (Top view with image, bottom view with images filtered out - just analytics), we see that the connections seem to open and close efficiently (also recall that images are downloading constantly while this data traffic is happening):

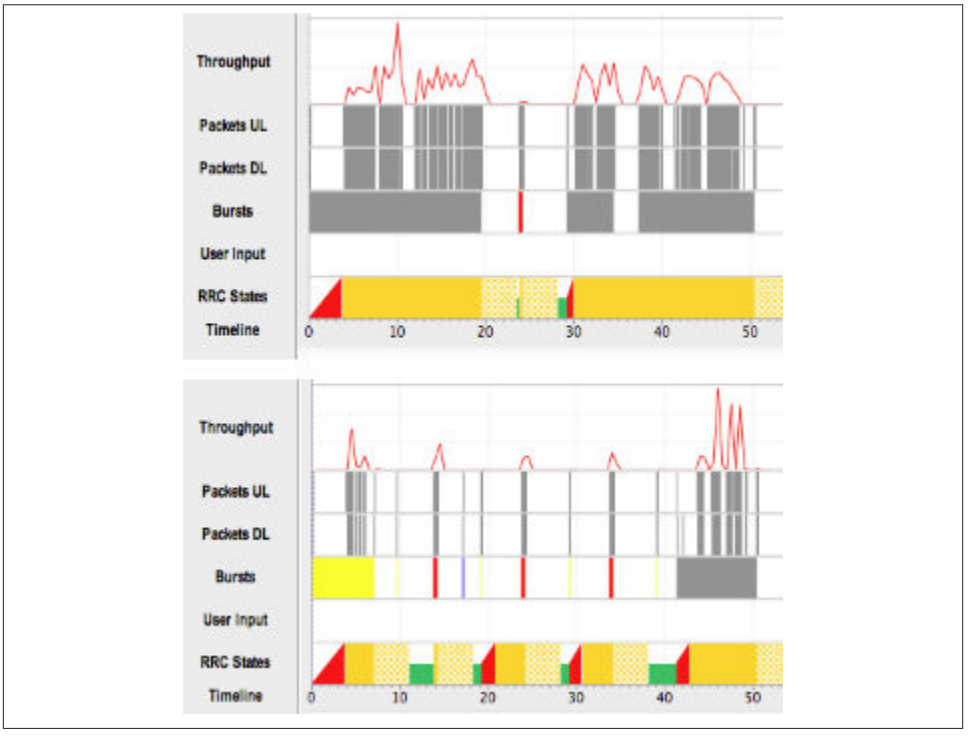


Figure 8-15. Network Trace of Image Scroll (Top) full app (Bottom) Just RUM connections

The RUM files from these providers are all encrypted, so to view them, I ran the same test through a "MITMProxy" on page 175. The RUM data being send by these anlytics providers does not endanger any customer data, and none fo the data being send 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 (see (to come) for more details.)

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 find 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 application 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 application with analytics, you can obtain very powerful data on how your application is failing, where it needs improvement and where things are running well with Real User Measurements. 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 applications 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 for slow website performance, and what little data I have seen for mobile applications follow the same trend. So, if you are having trouble convincing your management that application 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 sections of the book 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 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 application 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 he person testing and discovering issues.

Steve Souders' 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 less data usage, longer battery life and ultimately - this leads to happier customers. As a result, AT&T has instituted performance testing requirements for all internally developed applications, all applications that are pre-loaded on our devices, and we continue with outreach with developers both inside and outside the company.

Talking About Performance

In early 2011 (think middle 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 application 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 application, 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. Just a 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. Here are 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 application 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 multi MB 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 application. I wish I could say this is all automated, and or we use a stopwatch to time items. But 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 applications. By helping these developers - and showing the 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, and bugs and technical backlogs, it is tough), but correctly in 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 anew feature slows your app through your analytics. But, now all foxes are reactive to customers, since 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 application. 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 what the correct metrics for speed are There are many studies on what customers expect from the web (and there are more on apps that appear regularly). Do your own testing, and see what your users expect, and if they find your application 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 applications are so varied and unique that building a "go to" test case that holds for all applications 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 above are extremely generic, and could easily be tightened for a specific application. For your applications, 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 biggest complaint customers have about their Android phone 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 applications causing issues too. In Chapter 3 we looked at ways to measure the battery drain your application 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 application that are causing battery drain.

Your customers interact with your app on the screen, and and slow or janky scrolling is a prime factor in application 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 too simplify UI Hierarchies and test your UI for jank and speed issues with Systrace and other tools. If your application is suffering from crashes due to memory leaks or "not responding" issues, the tools discussed in Chapter 7 like MAT or Traceview 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 application consumes, to ensure that the experience always runs smoothly iIn 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. I discuss the often overlooked matter of security in [Link to Come], and started you on the pat of penetration testing of your application for security flaws. And finally, in Chapter 8, we looked at ways to get test results from your customers using Real User Measurement (RUM) tools. By understanding where your customer's pain points are, you can work backwards 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 application for performance gains. By digging into these tools and techniques, you will find issues that will speed up the rendering, performance and reduce the latency and battery drain of your application.