Version 0.4

for Android 3.2

Tuning Android Applications

Mark L. Murphy

CommonsWare

Tuning Android Applications

by Mark L. Murphy

COMMONSWARE

Tuning Android Applications
by Mark L. Murphy

Printing History:
Oct 2011: Version 0.4 ISBN: 978-0-9816780-6-1

Table of Contents

Welcome to the Warescription!

We hope you enjoy this ebook and its updates – subscribe to the Warescription newsletter on the Warescription site to learn when new editions of this book, or other books, are available.

All editions of CommonsWare titles, print and ebook, follow a software-style numbering system. Major releases (1.0, 2.0, etc.) are available in both print and ebook; minor releases (0.1, 0.9, etc.) are available in ebook form for Warescription subscribers only. Releases ending in .9 are "release candidates" for the next major release, lacking perhaps an index but otherwise being complete.

Each Warescription ebook is licensed for the exclusive use of its subscriber and is tagged with the subscriber's name. We ask that you not distribute these books. If you work for a firm and wish to have several employees have access, enterprise Warescriptions are available. Just contact us at enterprise@commonsware.com.

Also, bear in mind that eventually this edition of this title will be released under a Creative Commons license – more on this in the preface.

Remember that the CommonsWare Web site has errata and resources (e.g., source code) for each of our titles. Just visit the Web page for the book you are interested in and follow the links.

You can search through the PDF using most PDF readers (e.g., Adobe Reader). If you wish to search all of the CommonsWare books at once, and your operating system does not support that directly, you can always combine the PDFs into one, using tools like PDF Split-And-Merge or the Linux command `pdftk *.pdf cat output combined.pdf`.

Preface

Welcome to the Book!

Writing Android applications is not that hard. Writing Android applications that run crisply and stay within the confines of the limited capabilities of mobile devices is somewhat more difficult. This book aims to help you with these issues.

If you come to this book having read other CommonsWare Android books in the Warescription, thanks for sticking with the series! CommonsWare aims to have the most comprehensive set of Android development resources (outside of the Open Handset Alliance itself), and I appreciate your interest.

If you come to this book having learned about Android from other sources – or if you are new to Android entirely – thanks for joining the CommonsWare community! Android, while aimed at small devices, is a surprisingly vast platform, making it difficult for any given book, training, wiki, or other source to completely cover everything one needs to know.

And, most of all, thanks for your interest in this book! I sincerely hope you find it useful and at least occasionally entertaining.

What This Book Is All About

Mobile devices will always be compared to their higher powered brethren in desktops and notebooks. From a performance standpoint, therefore, mobile devices will lag behind for the foreseeable future. Mobile devices will have weaker CPUs, less RAM, less storage, tiny batteries, and so on.

Yet, users would prefer to not hear excuses as to why applications perform poorly or devices not deliver what they expect.

Hence, it is incumbent upon Android application developers to do what they can to make their apps work well given the limitations of your average mobile phone, tablet, and set-top box. This book aims to help developers identify where they have problems and how to address them.

This book is divided into five parts, for five major areas of performance tuning:

1. CPU, for application speed
2. Battery, to allow devices to run better longer
3. Memory, always a limitation compared to desktop applications
4. Bandwidth, a challenge given less-than-unlimited data plans, particularly ones that are metered
5. Storage, to wring more benefit out of limited space

Each of those parts has a short introductory chapter setting the stage, followed by two major themes:

1. How do you measure performance in this area and determine that you have a problem that needs solving? Where possible, we will explore how to narrow down the source of the poor performance as best as we can.
2. How do you fix these issues, from general purpose techniques (e.g., the NDK for compute-heavy algorithms) to targeted solutions (e.g., StrictMode for identifying where you are doing disk or network I/O on the main application thread). One chapter will focus on a major

technique for addressing the issue (e.g., NDK), and another chapter will cover other techniques.

Prerequisites

This book is for experienced Android developers. You should have written an app or two already, even if those apps were never distributed publicly. You should be comfortable with terms like "main application thread", "APK file", and "DDMS".

If you picked this book up expecting to learn Android application development from scratch, you really need another source first, since this book focuses on other topics. While we are fans of The Busy Coder's Guide to Android Development and the other books in the CommonsWare Warescription, there are plenty of other books available covering the Android basics, blog posts, wikis, and, of course, the main Android site itself.

Some chapters may reference material in previous chapters, though usually with a link back to the preceding section of relevance.

In order to make effective use of this book, you will want to download the source code for it off of the book's page on the CommonsWare site.

You can find out when new releases of this book are available via:

- The commonsguy Twitter feed
- The CommonsBlog
- The Warescription newsletter, which you can subscribe to off of your Warescription page

Getting Help

If you have questions about the book examples, visit StackOverflow and ask a question, tagged with **android** and **commonsware**.

If you have general Android developer questions, visit StackOverflow and ask a question, tagged with **android** (and any other relevant tags, such as **java**).

Warescription

This book will be published both in print and in digital form. The digital versions of all CommonsWare titles are available via an annual subscription – the Warescription.

The Warescription entitles you, for the duration of your subscription, to digital forms of *all* CommonsWare titles, not just the one you are reading. Presently, CommonsWare offers PDF and EPUB.

Each subscriber gets personalized editions of all editions of each title: both those mirroring printed editions and in-between updates that are only available in digital form. That way, your digital books are never out of date for long, and you can take advantage of new material as it is made available instead of having to wait for a whole new print edition. For example, when new releases of the Android SDK are made available, this book will be quickly updated to be accurate with changes in the APIs.

From time to time, subscribers will also receive access to subscriber-only online material, including not-yet-published new titles.

Also, if you own a print copy of a CommonsWare book, and it is in good clean condition with no marks or stickers, you can exchange that copy for a free four-month Warescription.

If you are interested in a Warescription, visit the Warescription section of the CommonsWare Web site.

Book Bug Bounty

Find a problem in one of our books? Let us know!

Be the first to report a unique concrete problem in the current digital edition, and we'll give you a coupon for a six-month Warescription as a bounty for helping us deliver a better product. You can use that coupon to get a new Warescription, renew an existing Warescription, or give the coupon to a friend, colleague, or some random person you meet on the subway.

By "concrete" problem, we mean things like:

- Typographical errors
- Sample applications that do not work as advertised, in the environment described in the book
- Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Each book has an errata page on the CommonsWare Web site; most known problems will be listed there. One coupon is given per email containing valid bug reports.

NOTE: Books with version numbers lower than 0.9 are ineligible for the bounty program, as they are in various stages of completion. We appreciate bug reports, though, if you choose to share them with us.

We appreciate hearing about "softer" issues as well, such as:

- Places where you think we are in error, but where we feel our interpretation is reasonable
- Places where you think we could add sample applications, or expand upon the existing material
- Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code

The source code samples shown in this book are available for download from a GitHub repository. All of the Android projects are licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

The Java projects are set up to be built by Ant or by Eclipse. If you wish to use the code with Eclipse, you should be able to simply import the project. If you wish to use Ant, run `android update project -p ...` (where ... is the path to a project of interest) on those projects you wish to use, so the build files are updated for your Android SDK version.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on July 1, **2015**. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-Share Alike 3.0 license, visit the Creative Commons Web site.

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Lifecycle of a CommonsWare Book

CommonsWare books generally go through a series of stages.

First are the pre-release editions. These will have version numbers below 0.9 (e.g., 0.2). These editions are incomplete, often times having but a few chapters to go along with outlines and notes. However, we make them available to those on the Warescription so they can get early access to the material.

Release candidates are editions with version numbers ending in ".9" (0.9, 1.9, etc.). These editions should be complete. Once again, they are made available to those on the Warescription so they get early access to the material and can file bug reports (and receive bounties in return!).

Major editions are those with version numbers ending in ".0" (1.0, 2.0, etc.). These will be first published digitally for the Warescription members, but will shortly thereafter be available in print from booksellers worldwide.

Versions between a major edition and the next release candidate (e.g., 1.1, 1.2) will contain bug fixes plus new material. Each of these editions should also be complete, in that you will not see any "TBD" (to be done) markers or the like. However, these editions may have bugs, and so bug reports are eligible for the bounty program, as with release candidates and major releases.

A book usually will progress fairly rapidly through the pre-release editions to the first release candidate and Version 1.0 – often times, only a few months. Depending on the book's scope, it may go through another cycle of significant improvement (versions 1.1 through 2.0), though this may take several months to a year or more. Eventually, though, the book will go into more of a "maintenance mode", only getting updates to fix bugs and deal with major ecosystem events – for example, a new release of the Android SDK will necessitate an update to all Android books.

PART I – CPU Performance

CHAPTER 1

Issues with Speed

Mobile devices are never fast enough. Either they are slow in general (e.g., slow CPU) or they are slow for particular operations (e.g., advanced game graphics).

What you do not want is for your application to be unnecessarily slow, where the user determines what is and is not "necessary". Your opinion of what is "necessary", alas, is of secondary importance.

This part of the book will focus on speed, including how you can measure and reduce lag in your applications. First, though, let's take a look at some of the specific issues surrounding speed.

Getting Things Done

In some cases, you simply cannot seem to get the work done that you want to accomplish. Your database query seems slow. Your encryption algorithm seems slow. Your image processing logic seems slow. And so on.

The limits of the device will certainly make this more of a problem than it might otherwise be. Even a current-era dual-core device will be slow compared to your average notebook or desktop. Also, this sort of speed issue is pervasive throughout computing, with decades of experience to help developers learn how to write leaner code.

This part of the book will aim to help you identify where the problem spots are, so you know what needs optimization, and then some Android-specific techniques for trying to improve matters.

Your UI Seems… Janky

Sometimes, the speed would be less of an issue for the user, if it was not freezing the UI or otherwise making it appear sluggish and "janky".

The Android widget framework operates in a single-threaded mode. All UI changes – from setting the text of a `TextView` to handling scrolling of a `GridView` – are processed as events on an event queue by the main application thread. That same thread is used for most UI callbacks, including activity lifecycle methods (e.g., `onCreate()`) and UI event methods (e.g., `onClick()` of a `Button`, `getView()` of an `Adapter`). Any time you take in those methods on the main application thread tie up that thread, preventing it from processing other GUI events or dispatching user input. For example, if your `getView()` processing in an `Adapter` takes too long, scrolling a `ListView` may appear slow compared to other `ListView` widgets in other applications.

Your objective is to identify where things are slow and move them into background operations. Some of this has been advised since the early days of Android, such as moving all network I/O to background threads. Some of this has arisen more recently, such as the move to use the "loader" framework to help you get data from data stores in the background for populating your UI.

This part of the book will point out ways for you to find out where you may be doing unfortunate things on the main application thread and techniques for getting that work handled by a background thread, or possibly eliminated outright.

Not Far Enough in the Background

Sometimes, even work you are trying to do in the background will seem to impact the foreground.

For example, you might think that your `Service` is automatically in the background. An `IntentService` does indeed use a background thread for processing commands via `onHandleIntent()`. However, all lifecycle methods of any `Service`, including `onStartCommand()`, are called on the main application thread. Hence, any time you take in those lifecycle methods will steal time away from GUI processing for the main application thread. The same holds true for `onReceive()` of a `BroadcastReceiver` and all the main methods of a `ContentProvider` (e.g., `query()`).

Even your background threads may not be sufficiently in the background. A process runs with a certain priority, using Linux process management APIs, based upon its state (e.g., if there is an activity in the foreground, it runs at a higher priority than if the process solely hosts some service). This will help to cap the CPU utilization of the background work, but only to a point. Similarly, threads that you fork – directly or via something like `IntentService` – may run at default priority rather than a lower priority. Even with lower priorities for the thread or process, every CPU instruction executed in the background is one clock tick that cannot be utilized by the foreground.

This part of the book will help you identify where you are taking lots of time on various threads and will help you manually manage priorities to help minimize the foreground impact of those threads, in addition to helping you reduce the amount of work those threads have to do.

Playing with Speed

Games, more so than most other applications, are highly speed-dependent. Everyone is seeking the "holy grail" of 60 frames per second (FPS) necessary for smooth animated effects. Not achieving that frame rate overall may mean the application will not appear quite as smooth; sporadically falling below that frame rate will result in jerky animation effects, much like the "janky" UIs in a non-game Android application.

For example, a classic problem with Android game development is garbage collection (GC). Only since the Gingerbread release of Android is the garbage collector concurrent, meaning that it runs in tandem with application code on a parallel thread. Historically, the Android garbage collector was a "stop the world" implementation, that would freeze the game long enough for a bit of GC work to be done before the game could continue. This behavior pretty much guaranteed sporadic failures to maintain a consistent frame rate. This caused game developers to have to take particular steps to avoid generating any garbage, such as maintaining its own object pools, to minimize or eliminate garbage collection pauses.

This book does not focus much on specific issues related to game development, though many of the techniques outlined here will be relevant for game developers.

CHAPTER 2

Finding CPU Bottlenecks

CPU issues tend to manifest themselves in three ways:

1. The user has a bad experience when using your app directly – scrolling is sluggish, activities take too long to display, etc.
2. The user has a bad experience when your app is running in the background, such as having slower frame rates on their favorite game because you are doing something complex in a service
3. The user has poor battery performance, driven by your excessive CPU utilization

Regardless of how the issue appears to the user, in the end, it is a matter of you using too much CPU time. That could be simply because your application is written to be constantly active (e.g., you have an everlasting service that uses `TimerTask` to wake up every second and do something). There is little anyone can do to help that short of totally rethinking the app's architecture (e.g., switch to `AlarmManager` and allow the user to configure the polling period).

However, in many cases, the problem is that you are using algorithms – yours or ones built into Android – that simply take too long when used improperly. This chapter will help you identify these bottlenecks, so you know what portions of your code need to be optimized in general or apply the techniques described in later chapters of this part of the book.

Traceview

The #1 tool in your toolbox for finding out where bottlenecks are occurring in your application is Traceview. This is available both within the Eclipse environment – though not as a separate perspective – and as a standalone tool.

What Is Traceview?

Traceview is Android's take on a method profiler. Profilers have existed for most other platforms, in one form or fashion, dating back to the mainframe days.

Technically, the profiling in Android is performed by the Dalvik virtual machine, under the direction of either DDMS or requests from your application code. Dalvik will write the "trace data" (call graphs showing methods, what they call, and the amount of time in each) to a file on external storage of the device or emulator. Traceview then views these trace files in a GUI, allowing you to visualize "hot spots", drill down to find where the time is being taken, and so forth.

At the time of this writing, Traceview is designed for use on single-core devices. Results on multi-core devices may be difficult to interpret.

Collecting Trace Data

Hence, the first step for finding where your CPU bottlenecks lie comes in the form of collecting trace data, to analyze with Traceview. As mentioned, there are two approaches for requesting trace data be logged: using the `Debug` class, and using DDMS.

Debug Class

If you know what chunk of code you want to profile, one way to arrange for the profile is to call `startMethodTracing()` on the `Debug` class. This takes the name of a trace file as a parameter and will begin recording all activity to that file, stored in the root of your external storage. You need to call `stopMethodTracing()` at some point to stop the trace – failing to do so will leave you with a corrupt trace file in the end.

Note that your application will need the `WRITE_EXTERNAL_STORAGE` permission for this to work. If your application does not normally need this permission, make yourself a note to remove it before you ship the production edition of your product, as there is no sense asking for any more permissions than you absolutely need. Also, your device or emulator will need enough external storage to hold the file, which can get very large for long traces – 100MB a minute is well within reason.

DDMS

Alternatively, you can initiate tracing via a toolbar button in DDMS. In both the DDMS perspective in Eclipse and the standalone DDMS, there is a button in the toolbar above the tree-table of devices and processes that toggles tracing on and off:

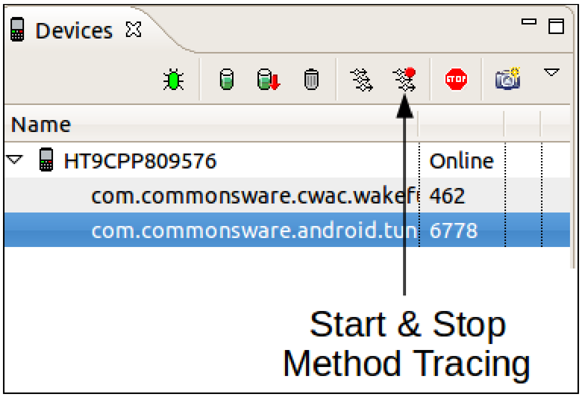


Figure 1. Toolbar button to start and stop method tracing

On Android 2.1 and earlier, this will write the trace out to a file on external storage, much as `startMethodTracing()` does. Hence, your application will need `WRITE_EXTERNAL_STORAGE` in this case, plus have enough external storage space to hold the file.

On Android 2.2 and newer, though, this data is written straight to the development machine, bypassing external storage. This means you do not need to worry about permissions or free space on your external storage. Hence, unless your problem only exists on Android 2.1 and earlier, you may find it easier to do your Traceview work on a newer Android device or emulator image. The file will wind up in your development machine's temporary directory (e.g., `/tmp` on Linux).

Performance While Tracing

Writing out each method invocation to a trace file adds significant overhead to your application. Run times can easily double or more. Hence, absolute times while tracing is enabled are largely meaningless – instead, as you analyze the data in Traceview, the goal is to examine relative times (i.e., such-and-so method takes up X% of the CPU time shown in the trace).

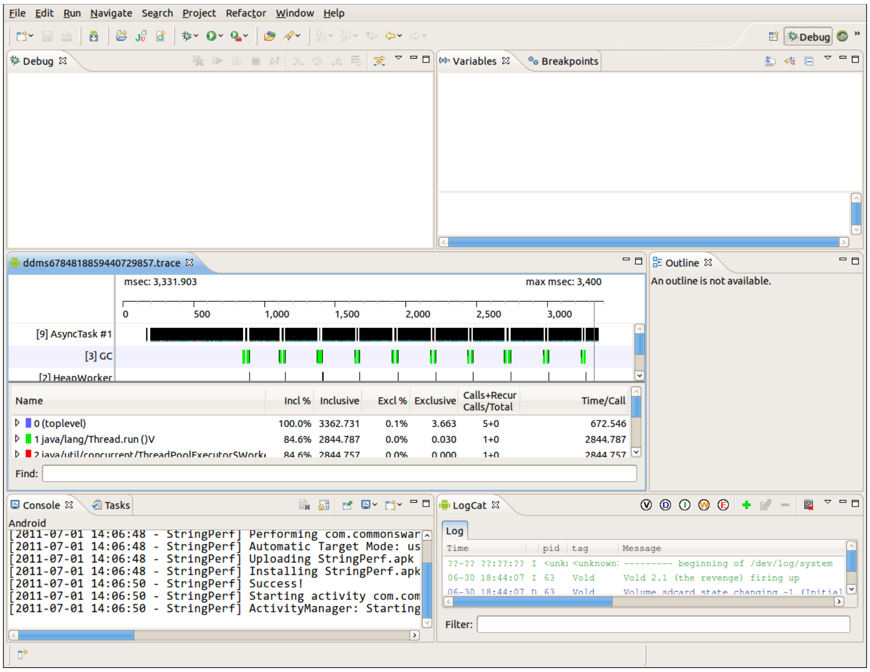
Also, running Traceview disables the JIT engine in Dalvik, further harming performance. Notably, this will not affect any native code you have added via the NDK, so an application run in Traceview will give you unusual results (much worse Java performance, more normal native performance).

Displaying Trace Data

Given that we have collected a trace file with data, the next step is to open up Traceview on that file. Depending on how you collected the file, Traceview may appear "automagically", or it may require you to manually start it up and point it to the trace file.

Eclipse/DDMS

If you used the DDMS perspective in Eclipse to record the trace data, the Debug perspective in Eclipse will automatically open up when you stop the tracing, showing you a Traceview tool:



Name	Incl %	Inclusive	Excl %	Exclusive	Calls+Recur Calls/Total	Time/Call
▷ 🟦 0 (toplevel)	100.0%	3362.731	0.1%	3.663	5+0	672.546
▷ 🟩 1 java/lang/Thread.run ()V	84.6%	2844.787	0.0%	0.030	1+0	2844.787
▷ 🟥 2 java/util/concurrent/ThreadPoolExecutor$Work	84.6%	2844.757	0.0%	0.000	1+0	2844.757

```
Android
[2011-07-01 14:06:48 - StringPerf] Performing com.commonswar
[2011-07-01 14:06:48 - StringPerf] Automatic Target Mode: us
[2011-07-01 14:06:48 - StringPerf] Uploading StringPerf.apk
[2011-07-01 14:06:48 - StringPerf] Installing StringPerf.apk
[2011-07-01 14:06:50 - StringPerf] Success!
[2011-07-01 14:06:50 - StringPerf] Starting activity com.com
[2011-07-01 14:06:50 - StringPerf] ActivityManager: Starting
```

```
Time            pid  tag        Message
??-?? ??:??:??  I <unk <unknown: --------- beginning of /dev/log/system
06-30 18:44:07  I 63  Vold       Vold 2.1 (the revenge) firing up
06-30 18:44:07  D 63  Vold       Volume sdcard state changing -1 (Initial
```

Figure 2. Debug perspective in Eclipse showing Traceview (middle left)

Standalone Traceview

If you used standalone DDMS and run a trace on Android 2.2 and up, it will automatically launch in the standalone Traceview utility.

If your trace file wound up on external storage on your device or emulator, you will need to download it to your development machine, whether using the File Manager within DDMS, or via the `adb pull` command. Once on your development machine, you can view it in the standalone Traceview tool using the `traceview` command:

```
traceview <path-to-trace-file>
```

Or, you can import the file into your Eclipse project, then double-click on it in the Project Explorer to view it in the Traceview tool.

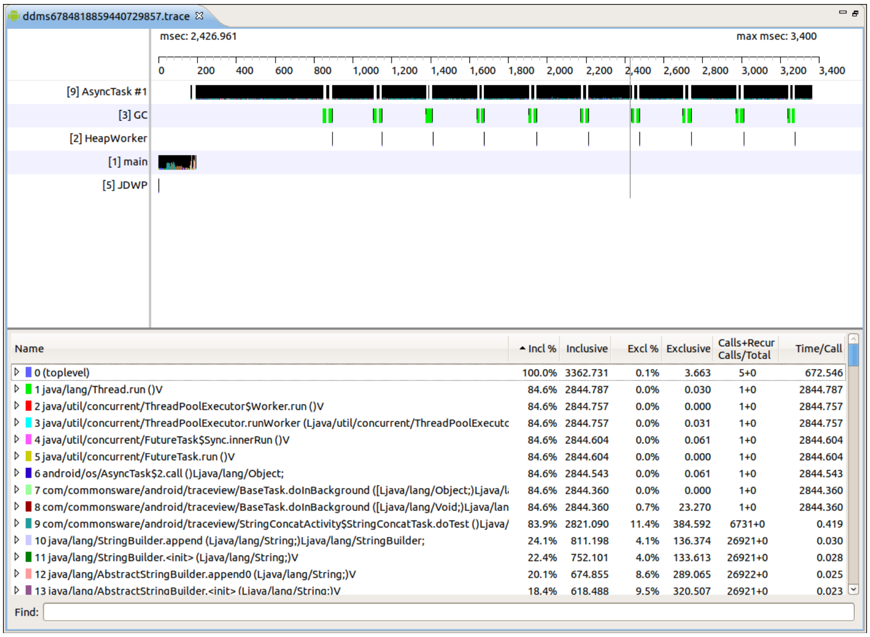
Interpreting Trace Data

Of course, the challenge is in making sense of what Traceview is trying to present.

For example, a classic performance bug in Java development is using string concatenation:

```
protected String doTest() {
  String result="This is a string";

  result+=" - that varies --";
  result+=" and also has ";
  result+=String.valueOf(4);
  result+=" hyphens in it";

  return(result);
}
```

Here is a Traceview screen showing that code executed 100,000 times, as packaged in a `StringPerfConcat` activity in the `Traceview` sample project in the book's source code:



Name	▲ Incl %	Inclusive	Excl %	Exclusive	Calls+Recur Calls/Total	Time/Call
0 (toplevel)	100.0%	3362.731	0.1%	3.663	5+0	672.546
1 java/lang/Thread.run ()V	84.6%	2844.787	0.0%	0.030	1+0	2844.787
2 java/util/concurrent/ThreadPoolExecutor$Worker.run ()V	84.6%	2844.757	0.0%	0.000	1+0	2844.757
3 java/util/concurrent/ThreadPoolExecutor.runWorker (Ljava/util/concurrent/ThreadPoolExecutc	84.6%	2844.757	0.0%	0.031	1+0	2844.757
4 java/util/concurrent/FutureTask$Sync.innerRun ()V	84.6%	2844.604	0.0%	0.061	1+0	2844.604
5 java/util/concurrent/FutureTask.run ()V	84.6%	2844.604	0.0%	0.000	1+0	2844.604
6 android/os/AsyncTask$2.call ()Ljava/lang/Object;	84.6%	2844.543	0.0%	0.061	1+0	2844.543
7 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Object;)Ljava/l	84.6%	2844.360	0.0%	0.000	1+0	2844.360
8 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Void;)Ljava/lan	84.6%	2844.360	0.7%	23.270	1+0	2844.360
9 com/commonsware/android/traceview/StringConcatActivity$StringConcatTask.doTest ()Ljava/	83.9%	2821.090	11.4%	384.592	6731+0	0.419
10 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;	24.1%	811.198	4.1%	136.374	26921+0	0.030
11 java/lang/StringBuilder.<init> (Ljava/lang/String;)V	22.4%	752.101	4.0%	133.613	26921+0	0.028
12 java/lang/AbstractStringBuilder.append0 (Ljava/lang/String;)V	20.1%	674.855	8.6%	289.065	26922+0	0.025
13 java/lang/AbstractStringBuilder.<init> (Ljava/lang/String;)V	18.4%	618.488	9.5%	320.507	26921+0	0.023

Figure 3. Expanded look at Traceview tool

The bars in the top portion of the display show different threads in the running application, in a timeline fashion, with time running from left to right. The "main" bar shows the main application thread, spending most of its time initializing the activity. The GC and HeapWorker threads are involved in garbage collection, popping in from time to time to collect garbage during 100,000 iterations of the above algorithm. Those 100,000 iterations are run in an `AsyncTask`, so we do not encounter an application-not-responding (ANR) dialog, and that is the "AsyncTask #1" thread at the top of the diagram.

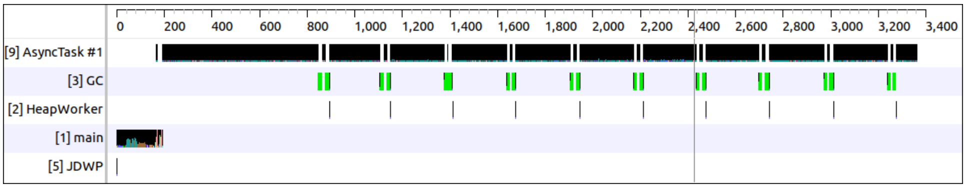
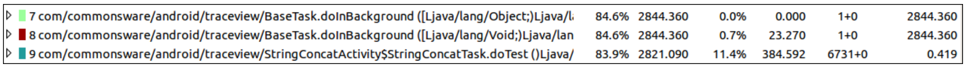


Figure 4. Zoomed in look at the TraceView thread timelines

You will notice that the horizontal timeline bars are not contiguous – there are gaps. In fact, if you were to combine all of the timelines into one, the "holes" in most of the rows would be filled by time in another row. This is illustrating that there is only one core on most Android device CPUs (these images were taken from a test run on a single-core Nexus One). We think of `AsyncTask` as moving work to the background, but it is important to remember that it still is consuming CPU time, even if the background thread means that we are not tying up the main application thread.

The bottom half of the display shows what methods are taking up all of the time, inclusively, in descending order. By "inclusively", Traceview means "code executed in this method and any methods it invokes". Hence, the top "100.0%" line shows the entry point to the whole application, and the next line shows where the `AsyncTask`'s background thread is being forked, and so on.

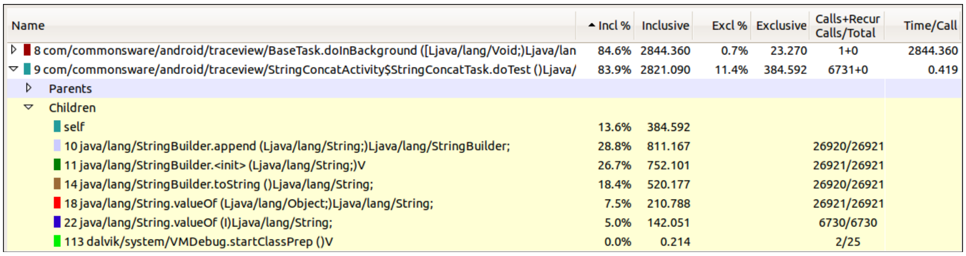
Typically, you want to find lines that reference your code. In this case, lines 7-9 are from the `com.commonsware` package. Let's focus on those:



7 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Object;)Ljava/l	84.6%	2844.360	0.0%	0.000	1+0	2844.360
8 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Void;)Ljava/lan	84.6%	2844.360	0.7%	23.270	1+0	2844.360
9 com/commonsware/android/traceview/StringConcatActivity$StringConcatTask.doTest ()Ljava/	83.9%	2821.090	11.4%	384.592	6731+0	0.419

Figure 5. Sample application method calls in Traceview

On their own, these lines are not especially informative. However, if we fold open the bottom row, using the arrow indicator on the left, we can drill down into what is going on inside that particular method, which happens to be the algorithm shown earlier in this section:



Name	Incl %	Inclusive	Excl %	Exclusive	Calls+Recur Calls/Total	Time/Call
8 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Void;)Ljava/lan	84.6%	2844.360	0.7%	23.270	1+0	2844.360
9 com/commonsware/android/traceview/StringConcatActivity$StringConcatTask.doTest ()Ljava/	83.9%	2821.090	11.4%	384.592	6731+0	0.419
Parents						
Children						
self	13.6%	384.592				
10 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;	28.8%	811.167			26920/26921	
11 java/lang/StringBuilder.<init> (Ljava/lang/String;)V	26.7%	752.101			26921/26921	
14 java/lang/StringBuilder.toString ()Ljava/lang/String;	18.4%	520.177			26920/26921	
18 java/lang/String.valueOf (Ljava/lang/Object;)Ljava/lang/String;	7.5%	210.788			26921/26921	
22 java/lang/String.valueOf (I)Ljava/lang/String;	5.0%	142.051			6730/6730	
113 dalvik/system/VMDebug.startClassPrep ()V	0.0%	0.214			2/25	

Figure 6. Drilling down in Traceview

The "self" line refers to code that is directly executed in the method, not involving a nested method call, such as variable declarations and returning values. We see the `valueOf()` calls, along with three rows showing references to `StringBuilder`. On the surface, that may seem odd, considering that we are not referring to `StringBuilder` in the source code.

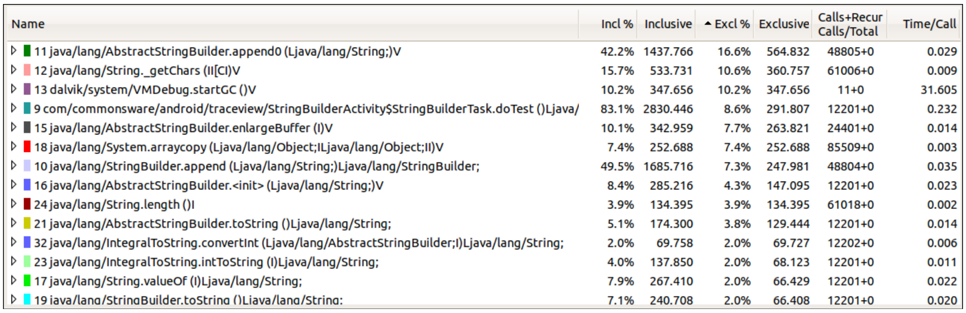
It turns out that the `javac` compiler replaces string concatenation with `append()` calls on a `StringBuilder`, created on the fly for that specific concatenation. So, of the 83.9% of the time taken up in the entire run by the `doTest()` method, 26.7% is taken up by creating these temporary `StringBuilder` objects, 28.8% is consumed by calling `append()` on the `StringBuilder`, and another 18.4% is used by calling `toString()` to get the resulting `String` out of the `StringBuilder`.

This suggests an optimization: we could create our own `StringBuilder` and use it for concatenating the text, thereby saving us creating a few temporary ones and calling `toString()` extra times:

```
protected String doTest() {
  StringBuilder result=new StringBuilder("This is a string");

  result.append(" - that varies --");
  result.append(" and also has ");
  result.append(String.valueOf(4));
  result.append(" hyphens in it");

  return(result.toString());
}
```

This implementation of the algorithm runs about twice as fast as the first.

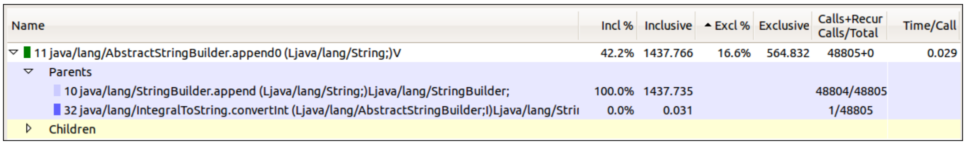
The "Exclusive" and "Excl %" columns show how much time is taken in an individual method itself, not including any children. If you sort on that, you see the specific local spots where time is being taken up. For example, here is a Traceview roster from testing the second algorithm shown above (the `StringPerfBuilder` activity):



Name	Incl %	Inclusive	▲ Excl %	Exclusive	Calls+Recur Calls/Total	Time/Call
▷ 🟩 11 java/lang/AbstractStringBuilder.append0 (Ljava/lang/String;)V	42.2%	1437.766	16.6%	564.832	48805+0	0.029
▷ 🟥 12 java/lang/String._getChars (II[CI)V	15.7%	533.731	10.6%	360.757	61006+0	0.009
▷ 🟪 13 dalvik/system/VMDebug.startGC ()V	10.2%	347.656	10.2%	347.656	11+0	31.605
▷ 🟦 9 com/commonsware/android/traceview/StringBuilderActivity$StringBuilderTask.doTest ()Ljava/	83.1%	2830.446	8.6%	291.807	12201+0	0.232
▷ ⬛ 15 java/lang/AbstractStringBuilder.enlargeBuffer (I)V	10.1%	342.959	7.7%	263.821	24401+0	0.014
▷ 🟥 18 java/lang/System.arraycopy (Ljava/lang/Object;ILjava/lang/Object;II)V	7.4%	252.688	7.4%	252.688	85509+0	0.003
▷ ⬜ 10 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;	49.5%	1685.716	7.3%	247.981	48804+0	0.035
▷ 🟦 16 java/lang/AbstractStringBuilder.<init> (Ljava/lang/String;)V	8.4%	285.216	4.3%	147.095	12201+0	0.023
▷ 🟥 24 java/lang/String.length ()I	3.9%	134.395	3.9%	134.395	61018+0	0.002
▷ 🟨 21 java/lang/AbstractStringBuilder.toString ()Ljava/lang/String;	5.1%	174.300	3.8%	129.444	12201+0	0.014
▷ 🟦 32 java/lang/IntegralToString.convertInt (Ljava/lang/AbstractStringBuilder;I)Ljava/lang/String;	2.0%	69.758	2.0%	69.727	12202+0	0.006
▷ 🟩 23 java/lang/IntegralToString.intToString (I)Ljava/lang/String;	4.0%	137.850	2.0%	68.123	12201+0	0.011
▷ 🟩 17 java/lang/String.valueOf (I)Ljava/lang/String;	7.9%	267.410	2.0%	66.429	12201+0	0.022
▷ 🟦 19 java/lang/StringBuilder.toString ()Ljava/lang/String;	7.1%	240.708	2.0%	66.408	12201+0	0.020

Figure 7. Traceview, sorted by exclusive time

We see that the top three culprits are all Android/Dalvik methods, which we cannot optimize. Instead, the fact that they are taking up so much time is indicative of the fact that we are calling them a lot, also in evidence by the Calls/Total column. You can examine the parents of a call to see where those calls come from, to see if you can change upstream code to result in fewer such calls:



Name	Incl %	Inclusive	▲ Excl %	Exclusive	Calls+Recur Calls/Total	Time/Call
▽ 🟩 11 java/lang/AbstractStringBuilder.append0 (Ljava/lang/String;)V	42.2%	1437.766	16.6%	564.832	48805+0	0.029
▽ Parents						
⬜ 10 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder;	100.0%	1437.735			48804/48805	
🟦 32 java/lang/IntegralToString.convertInt (Ljava/lang/AbstractStringBuilder;I)Ljava/lang/Stri	0.0%	0.031			1/48805	
▷ Children						

Figure 8. Traceview, showing parents of a method call

Here, we can see that all those `append0()` calls are triggered by calls to `append()` on the `StringBuilder`, which is not terribly surprising.

You can also zoom in to take a very narrow look at the data. Simply click-drag a bar in the timeline to select an region to zoom into. The timeline will switch to show just that range of milliseconds and the calls that take place there:

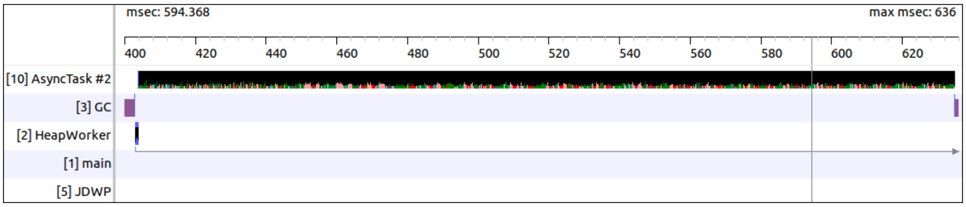
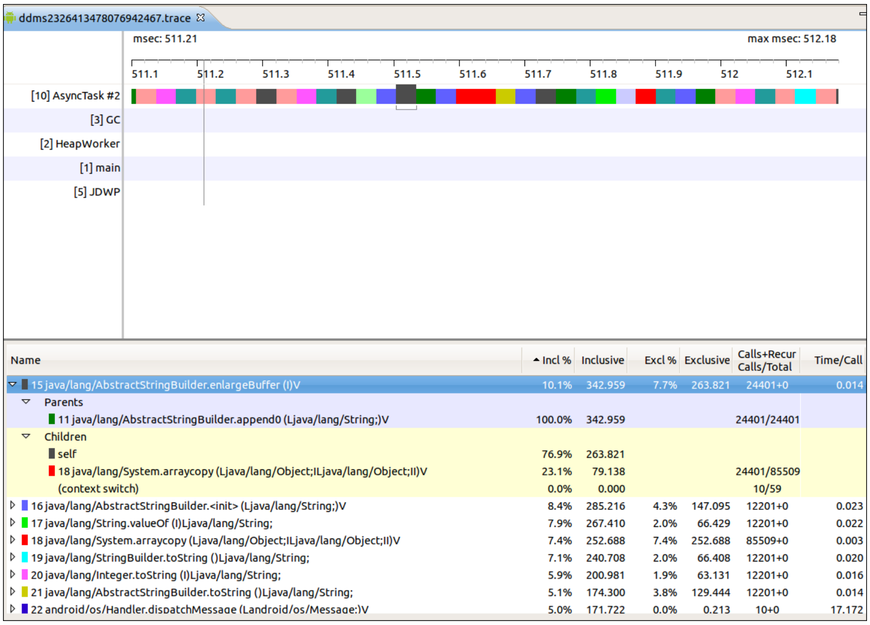


Figure 9. Traceview, zoomed in on ~230 milliseconds of run time

If you zoom in far enough, you will start seeing solid blocks of color, corresponding to the color-coded methods in the table of results on the bottom half of the screen. You can tap on any block of color to bring up that specific method in the table:



Name	Incl %	Inclusive	Excl %	Exclusive	Calls+Recur Calls/Total	Time/Call
▽ ■ 15 java/lang/AbstractStringBuilder.enlargeBuffer (I)V	10.1%	342.959	7.7%	263.821	24401+0	0.014
▽ Parents						
🟩 11 java/lang/AbstractStringBuilder.append0 (Ljava/lang/String;)V	100.0%	342.959			24401/24401	
▽ Children						
■ self	76.9%	263.821				
🟥 18 java/lang/System.arraycopy (Ljava/lang/Object;ILjava/lang/Object;II)V	23.1%	79.138			24401/85509	
(context switch)	0.0%	0.000			10/59	
▷ 🟦 16 java/lang/AbstractStringBuilder.<init> (Ljava/lang/String;)V	8.4%	285.216	4.3%	147.095	12201+0	0.023
▷ 🟩 17 java/lang/String.valueOf (I)Ljava/lang/String;	7.9%	267.410	2.0%	66.429	12201+0	0.022
▷ 🟥 18 java/lang/System.arraycopy (Ljava/lang/Object;ILjava/lang/Object;II)V	7.4%	252.688	7.4%	252.688	85509+0	0.003
▷ 🟦 19 java/lang/StringBuilder.toString ()Ljava/lang/String;	7.1%	240.708	2.0%	66.408	12201+0	0.020
▷ 🟪 20 java/lang/Integer.toString (I)Ljava/lang/String;	5.9%	200.981	1.9%	63.131	12201+0	0.016
▷ 🟨 21 java/lang/AbstractStringBuilder.toString ()Ljava/lang/String;	5.1%	174.300	3.8%	129.444	12201+0	0.014
▷ 🟦 22 android/os/Handler.dispatchMessage (Landroid/os/Message;)V	5.0%	171.722	0.0%	0.213	10+0	17.172

Figure 10. Traceview, zoomed in on ~1 millisecond of run time, highlighting one specific method

Zooming back out, though, is somewhat of a pain. If you drag the timeline itself (not one of the bars, but the "meter stick" showing the milliseconds) from left to right, you will zoom out. Do this enough times, and you can return approximately to the original state.

Other General CPU Measurement Techniques

While Traceview is great for narrowing down a general performance issue to a specific portion of code, it does assume that you know approximately where the problem is, or that you even have a problem in the first place. There are other approaches to help you identify if and (roughly) where you have problems, which you can then attack with Traceview to try to refine.

Logging

Traceview can be useful, if you have a rough idea of where your performance problem lies and need to narrow it down further. If you have a large and complicated application, though, trying to sift through all of it in Traceview may be difficult.

However, there is nothing stopping you from using good old-fashioned logging to get a rough idea of where your problems lie, for further analysis via Traceview. Just sprinkle your code with `Log.d()` calls, logging `SystemClock.uptimeMillis()` with an appropriate label to identify where you were at that moment in time. "Eyeballing" the LogCat output can illustrate areas where unexpected delays are occurring – the areas in which you can focus more time using Traceview.

A useful utility class for this is `TimingLogger`, in the `android.util` package. It will collect a series of "splits" and can dump them to LogCat along with the overall time between the creation of the `TimingLogger` object and the corresponding `dumpToLog()` method call. Note, though, that this will only log to LogCat when you call `dumpToLog()` – all of the calls to `split()` to record intermediate times have their results buffered until `dumpToLog()` is called. Also note that logging needs to be set to VERBOSE for this information to actually be logged – use the command `adb shell setprop log.tag.LOG_TAG VERBOSE`, substituting your log tag (supplied to the `TimingLogger` constructor) for `LOG_TAG`.

FPS Calculations

Sometimes, it may not even be strictly obvious how bad the problem is. For example, consider scrolling a `ListView`. Some performance issues, like sporadic "hiccups" in the scrolling, will be visually apparent. However, absent those, it may be difficult to determine whether your particular `ListView` is behaving more slowly than you would expect.

A classic measurement for games is frames per second (FPS). Game developers aim for a high FPS value – 60 FPS is considered to be fairly smooth, for example. However, this sort of calculation can only really be done for applications that are continuously drawing – such as Romain Guy's WindowBackground sample application. Ordinary Android widget-based UIs are only drawing based upon user interaction or, possibly, upon background updates to data. In other words, if the UI will not even be trying to draw 60 times in a second, trying to measure FPS to get 60 FPS is pointless.

You may be able to achieve similar results, though, simply by logging how long it takes to, say, fling a list (use `setOnScrollListener()` and watch for `SCROLL_STATE_FLING` and other events).

StrictMode for Main Application Thread Issues

NOTE: the following material is excerpted from The Busy Coder's Guide to Android Development, which has additional coverage of `StrictMode`.

Users are more likely to like your application if, to them, it feels responsive. Here, by "responsive", we mean that it reacts swiftly and accurately to user operations, like taps and swipes.

Conversely, users are less likely to be happy with you if they perceive that your UI is "janky" – sluggish to respond to their requests. For example, maybe your lists do not scroll as smoothly as they would like, or tapping a button does not yield the immediate results they seek.

While threads and `AsyncTask` and the like can help, it may not always be obvious where you should be applying them. However, there are a few standard sorts of things that developers do, sometimes quite by accident, on the main application thread that will tend to cause sluggishness:

- Flash I/O, both for the on-board storage and for "external storage" (e.g., the SD card)
- Network I/O

However, even here, it may not be obvious that you are performing these operations on the main application thread. This is particularly true when the operations are really being done by Android's code that you are simply calling.

That is where `StrictMode` comes in. Its mission is to help you determine when you are doing things on the main application thread that might cause a janky user experience.

Setting up Strict Mode

`StrictMode` works on a set of policies. There are presently two categories of policies: VM policies and thread policies. The former represent bad coding practices that pertain to your entire application, notably leaking SQLite `Cursor` objects and kin. The latter represent things that are bad when performed on the main application thread, notably flash I/O and network I/O.

Each policy dictates what `StrictMode` should watch for (e.g., flash reads are OK but flash writes are not) and how `StrictMode` should react when you violate the rules, such as:

- Log a message to LogCat
- Display a dialog
- Crash your application (seriously!)

The simplest thing to do is call the static `enableDefaults()` method on `StrictMode` from `onCreate()` of your first activity. This will set up normal operation, reporting all violations by simply logging to LogCat. However, you can set your own custom policies via `Builder` objects if you so choose.

Development Only, Please!

Do not use `StrictMode` in production code. It is designed for use when you are building, testing, and debugging your application. It is not designed to be used in the field.

To deal with this, you could:

- Simply comment out or remove the `StrictMode` setup code when you prepare your production builds
- Use some sort of production flag to skip the `StrictMode` setup code when needed

CHAPTER 3

Focus On: NDK

When Android was first released, many a developer wanted to run C/C++ code on it. There was little support for this, other than by distributing a binary executable and running it via a forked process. While this works, it is a bit cumbersome, and the process-based interface limits how cleanly your C/C++ code could interact with a Java-based UI. On top of all of that, the use of such binary executables is not well supported.

In June 2009, the core Android team released the Native Development Kit (NDK). This allows developers to write C/C++ for Android applications in a supported fashion, in the form of libraries linked to a hosting Java-based application via the Java Native Interface (JNI). This offers a wealth of opportunities for Android development, and this part of the book will explore how you can take advantage of the NDK to exploit those opportunities.

This chapter explains how to set up the NDK and apply it to your project. What it does not do is attempt to cover all possible uses of the NDK – game applications in particular have access to many frameworks, like OpenGL and OpenSL, that are beyond the scope of this book.

The Role of the NDK

We start by examining Dalvik's primarily limitation – speed. Next, we look at the reasons one might choose the NDK, speed among them. We wrap up with some reasons why the NDK may not be the right solution for every Android problem, despite its benefits.

Dalvik: Secure, Yes; Speedy, Not So Much

Dalvik was written with security as a high priority. Android's security architecture is built around Linux's user model, with each application getting its own user ID. With each application's process running under its own user ID, one process cannot readily affect other processes, helping to contain any single security flaw in an Android application or subsystem. This requires a fair number of processes. However, phones have limited RAM, and the Android project wanted to offer Java-based development. Multiple processes hosting their own Java virtual machines simply could not fit in a phone. Dalvik's virtual machine is designed to address this, maximizing the amount of the virtual machine that can be shared securely between processes (e.g., via "copy-on-write").

Of course, it is wonderful that Android has security so woven into the fabric of its implementation. However, inventing a new virtual machine required tradeoffs, and most of those are related to speed.

A fair amount of work has gone into making Java fast. Standard Java virtual machines do a remarkable job of optimizing applications on the fly, such that Java applications can perform at speeds near their C/C++ counterparts. This borders on the amazing and is a testament to the many engineers who put countless years into Java.

Dalvik, by comparison, is very young. Many of Java's performance optimization techniques – such as advanced garbage collection algorithms – simply have not been implemented to nearly the same level in Dalvik. This is not to say they will never exist, but it will take some time. Even then, though, there may be limits as to how fast Dalvik can operate, considering that it cannot "throw memory at the problem" to the extent Java can on the desktop or server.

If you need speed, Dalvik is not the answer today, and may not be the answer tomorrow, either.

Going Native

Java-based Android development via Dalvik and the Android SDK is far and away the option with the best support from the core Android team. HTML5 application development is another option that was brought to you by the core Android development team. The third leg of the official Android development triad is the NDK, provided to developers to address some specific problems, outlined below.

Speed

Far and away the biggest reason for using the NDK is speed, pure and simple. Writing in C/C++ for the device's CPU will be a major speed improvement over writing the same algorithms in Java, despite Android's just-in-time (JIT) compiler.

There is overhead in reaching out to the C/C++ code from a hosting Java application, and so for the best performance, you will want a coarse interface, without a lot of calls back and forth between Java and the native opcodes. This may require some redesign of what might otherwise be the "natural" way of writing the C/C++ code, or you may just have to settle for less of a speed improvement. Regardless, for many types of algorithms – from cryptography to game AI to video format conversions – using C/C++ with the NDK will make your application perform much better, to the point where it can enable applications to be successful that would be entirely too slow if written solely in Java.

Bear in mind, though, that much of what you think is Java code in your app really is native "under the covers". Many of the built-in Android classes are thin shims over native implementations. Again, focus on applying the NDK where you are performing lots of work yourself in Java code that might benefit from the performance gains.

Porting

You may already have some C/C++ code, written for another environment, that you would like to use with Android. That might be for a desktop application. That might be for another mobile platform, such as iPhone or WebOS, where C/C++ is an option. That might be for mobile platform, such as Symbian, where C/C++ is the conventional solution, rather than some other language. Regardless, so long as that code is itself relatively platform-independent, it should be usable on Android.

This may significantly streamline your ability to support multiple platforms for your application, even if down-to-the-metal speed is not really something you necessarily need. This may also allow you to reuse existing C/C++ code written by others, for image processing or scripting languages or anything else.

Knowing Your Limits

Developers love silver bullets. Developers are forevermore seeking The One True Approach to development that will be problem-free. Sisyphus would approve, of course, as development always involves tradeoffs. So while the NDK's speed may make it tantalizing, it is not a solution for general Android application development, for several reasons, explored in this section.

Android APIs

The biggest issue with the NDK is that you have very limited access to Android itself. There are a few libraries bundled with Android that you can leverage, and a few other APIs offered specifically to the NDK, such as the ability to render OpenGL 3D graphics. But, generally speaking, the NDK has no access to the Android SDK, except by way of objects made available to it from the hosting application via JNI.

As such, it is best to view the NDK as a way of speeding up particular pieces of an SDK application – game physics, audio processing, OCR, and the like.

All of those are algorithms that need to run on Android devices with data obtained from Android, but otherwise are independent of Android itself.

Cross-Platform Compatibility

While C/C++ *can* be written for cross-platform use, often it is not.

Sometimes, the disparity is one of APIs. Any time you use an API from a platform (e.g., iPhone) or a library (e.g., Qt) not available on Android, you introduce an incompatibility. This means that while a lot of your code – measured in terms of lines – may be fine for Android, there may be enough platform-specific bits woven throughout it that you would have a significant rewrite ahead of you to make it truly cross-platform.

Android itself, though, has a compatibility issue, in terms of CPUs. Android mostly runs on ARM devices today, since Android's initial focus was on smartphones, and ARM-powered smartphones at that. However, the focus on ARM will continue to waver, particularly as Android moves into other devices where other CPU architectures are more prevalent, such as Atom or MIPS for set-top boxes. While your code may be written in a fashion that works on all those architectures, the binaries that code produces will be specific to one architecture. The NDK gives you additional assistance in managing that, so that your application can simultaneously support multiple architectures. Right now, the r6 version of the NDK is for ARM and x86.

Maturity

The Dalvik VM is young. The NDK is younger still, debuting in mid-2009. Fewer developers have been using the NDK than have been using the SDK. The combination of age and usage gives the NDK a fairly short track record, meaning that there may be more NDK problems than are presently known.

Available Expertise

If you are seeking outside assistance for your Android development efforts, there will be fewer people available to assist you with NDK development, compared to SDK development. The NDK is newer than the SDK, so many developers started with what was originally available. Many applications do not need the NDK, and so many developers will not have taken the time to learn how to use it. Furthermore, many Android developers may be far more fluent in Java than they are in C/C++, based on their own backgrounds, and so they would tend to stick with tools they are more comfortable with. To top it off, few books on Android development cover the NDK, though this is being incrementally improved, via books such as this one.

If you are looking for somebody with NDK experience, *ask for it* – do not assume that Android developers know the NDK nearly as well as they know the SDK.

NDK Installation and Project Setup

The Android NDK is blissfully easy to install, in some ways even easier than is the Android SDK. Similarly, setting up an NDK-equipped project is rather straightforward. However, the documentation for the NDK is mostly a set of text files (`OVERVIEW.TXT` prominent among them). These are well-written but suffer from the limits of the plain-text form factor, plus are focused strictly on the NDK and not the larger issue of Android projects that use the NDK.

This chapter will fill in some of those gaps.

Installing the NDK

As with the Android SDK, the Android NDK comes in the form of a ZIP file, containing everything you need to build NDK-enabled Android applications. Hence, setting up the NDK is fairly trivial, particularly if you are developing on Linux.

Prerequisites

You will need the GNU `make` and GNU `awk` packages installed. These may be part of your environment already. For example, in Ubuntu, run `sudo aptitude install make gawk`, or use the Synaptic Package Manager, to ensure you have these two packages.

While you can do NDK development directly on Linux or OS X, NDK development on Windows can only be done using the Cygwin environment. This gives you a Linux-style shell and Linux-style tools on a Windows PC. In addition to a base Cygwin 1.7 (or newer) installation, you will need the make and gawk Cygwin packages installed in Cygwin.

If you encounter difficulties with Cygwin, you may wish to consider whether running Linux in a virtualization environment (e.g., VirtualBox) might be a better solution for you.

Download and Unpack

The Android NDK per-platform (Linux/OS X/Windows) ZIP files can be downloaded from the NDK page on the Android Developers site. These ZIP files are not small (~50MB each), because they contain the entire toolchain – that is why there are so few prerequisites.

You are welcome to unpack the ZIP file anywhere it makes sense on your development machine. However, putting it *inside* the Android SDK directory may not be a wise move – a peer directory would be a safer choice. You are welcome to rename the directory if you choose.

Environment Variables

The NDK documentation will cite an `NDK` environment variable, set to point to the directory in which you unpacked the NDK. This is a documentation convention and does not appear to be required for actual use of the NDK, though it is not a bad idea. You could also consider adding the NDK directory to your `PATH`, though that too is not required.

Bear in mind that you will be using the NDK tools from the command line, and so being able to conveniently reference this directory is reasonably important.

Setting Up an NDK Project

At its core, an NDK-enhanced Android project is a regular Android project. You still need a manifest, layouts, Java source code, and all the other trappings of a regular Android application. The NDK simply enables you to add C/C++ code to that project and have it included in your builds, referenced from your Java code via the Java Native Interface (JNI).

The examples shown in this section are from the `JNI/WeakBench` project, which implements a pair of benchmarks in Java and C, to help demonstrate the performance differences between the environments.

Writing Your C/C++ Code

The first step towards adding NDK code to your project is to create a `jni/` directory and place your C/C++ code inside of it. While there are ways to use a different base directory, it is unclear why you would need to. How you organize the code inside of `jni/` is up to you. C++ code should use `.cpp` as file extensions, though this too is configurable.

Your C/C++ code will be made up of two facets:

1. The code doing the real work
2. The code implementing your JNI interface

If you have never used JNI before, JNI uses naming conventions to tie functions in a C/C++ library to their corresponding hooks in the Java code.

For example, in the `WeakBench` project in the book's source code, you will find `jni/weakbench.c`:

```
#include <stdlib.h>
#include <math.h>
#include <jni.h>

typedef unsigned char boolean;

static void nsieve(int m) {
 unsigned int count = 0, i, j;
 boolean * flags = (boolean *) malloc(m * sizeof(boolean));
 memset(flags, 1, m);

 for (i = 2; i < m; ++i)
 if (flags[i]) {
 ++count;
 for (j = i << 1; j < m; j += i)
// if (flags[j])
 flags[j] = 0;
 }

 free(flags);
}

void
Java_com_commonsware_android_tuning_weakbench_WeakBench_nsievenative( JNIEnv*
env,
 jobject thiz )
{
 int i=0;
 for (i = 0; i < 3; i++)
 nsieve(10000 << (9-i));
}

double eval_A(int i, int j) { return 1.0/((i+j)*(i+j+1)/2+i+1); }

void eval_A_times_u(int N, const double u[], double Au[])
{
 int i,j;
 for(i=0;i<N;i++)
 {
 Au[i]=0;
 for(j=0;j<N;j++) Au[i]+=eval_A(i,j)*u[j];
 }
}

void eval_At_times_u(int N, const double u[], double Au[])
{
 int i,j;
 for(i=0;i<N;i++)
 {
 Au[i]=0;
 for(j=0;j<N;j++) Au[i]+=eval_A(j,i)*u[j];
 }
}
```

```
void eval_AtA_times_u(int N, const double u[], double AtAu[])
{ double v[N]; eval_A_times_u(N,u,v); eval_At_times_u(N,v,AtAu); }


void
Java_com_commonsware_android_tuning_weakbench_WeakBench_specnative( JNIEnv* env,
 jobject thiz )
{
 int i;
 int N = 1000;
 double u[N],v[N],vBv,vv;
 for(i=0;i<N;i++) u[i]=1;
 for(i=0;i<10;i++)
 {
 eval_AtA_times_u(N,u,v);
 eval_AtA_times_u(N,v,u);
 }
 vBv=vv=0;
 for(i=0;i<N;i++) { vBv+=u[i]*v[i]; vv+=v[i]*v[i]; }
}
```

Much of the code shown here comes from the Great Language Benchmarks Game, specifically their `nsieve` and `spectral-norm` benchmarks. And, much of the code looks like normal C code.

Two functions, though, serve as JNI entry points:

1. `Java_com_commonsware_abj_weakbench_WeakBench_nsievenative`
2. `Java_com_commonsware_abj_weakbench_WeakBench_specnative`

As will be seen later in this section, these will map to `nsievenative()` and `specnative()` methods on a `com.commonsware.abj.weakbench.WeakBench` class. The Java class (with package) and method names are converted into a function call name, so JNI can identify the function at runtime.

The implementation of these methods do not make use of any Java objects, nor do they return anything – they just implement the benchmark. A slightly more elaborate JNI API will be demonstrated later in this chapter.

Writing Your Makefile(s)

To tell the NDK tools how to build your code, you will need one or two makefiles.

Building Your Library

Any time you modify your C/C++ code, or the makefiles, you will need to build your NDK library. To do that, from a command prompt in your project's root directory, run the `ndk-build` script found in the NDK's root directory. In other words, if you set up an NDK environment variable to point to where you have the NDK installed, execute `$NDK/ndk-build` from your project root.

This will compile and link your C/C++ code into a library (or conceivably several libraries, if you have a complex set of `Android.mk` files). These will wind up in your project's `libs/` directory, in subdirectories based on your CPU architectures indicated by your `Application.mk` file.

For example, if you run `$NDK/ndk-build` from the `WeakBench` project root, you will wind up with a `libs/armeabi/libweakbench.so` file. The `armeabi` portion is because that is the default CPU architecture that the NDK supports, and `WeakBench` did not change the defaults via an `Application.mk` file. The "weakbench" portion of `libweakbench.so` is because our `LOCAL_MODULE` value in our `Android.mk` file is `weakbench`. The `lib` prefix is automatically added by the build tools. The `.so` file extension is because our `Android.mk` file indicated that we are building a shared library (via the `BUILD_SHARED_LIBRARY` directive), and `.so` is the standard file extension for shared libraries in Linux (and, hence, Android).

You are welcome to add this to your build process, such as adding it to your Ant build script, though it is not automatically included in the build process as defined by Android.

Using Your Library Via JNI

Now that you have your base C/C++ code being successfully compiled by the NDK, you need to turn your attention towards crafting the bridge between the Dalvik VM and the C/C++ code, following in the conventions of the Java Native Interface (JNI).

This section, while explaining the various steps involved in using the JNI, is far from a complete treatise on the subject. If you are going to spend a lot of time working with JNI, you are encouraged to seek additional resources on this topic, such as Core Java: Volume II, which has a chapter on JNI.

We created two C functions for accessing benchmarks:

1. `Java_com_commonsware_abj_weakbench_WeakBench_nsievenative`
2. `Java_com_commonsware_abj_weakbench_WeakBench_specnative`

Those, in turn, need to be defined as static methods on a `com.commonsware.abj.weakbench.WeakBench` class. Moreover, these methods will need to have the `native` keyword, indicating that their implementation is not found in Java code, but in native C/C++ code. The naming convention of the C functions allows the Dalvik runtime to identify what function names should be used for those `native` method implementations.

However, that alone will be insufficient – we need to tell Dalvik where it can find the library in the first place. While naming conventions are good enough for the C function names, there is no corresponding naming convention for the library itself.

To do this, we use the `loadLibrary()` static method on the `System` class. A class implementing native methods should call `loadLibrary()` in a `static` block, so it is executed when the class is first referenced. For the NDK, all we need to do is supply the name we gave the library in the `Android.mk` file.

Here is the portion of the `WeakBench` class that has the `native` methods and the `loadLibrary()` call:

```
static {
  System.loadLibrary("weakbench");
}

public native void nsievenative();
public native void specnative();
```

Now, we can call our `nsievenative()` and `specnative()` methods on `WeakBench`, just as if they were regular Dalvik methods on a regular Dalvik class. The fact that they are really going off and invoking C functions is purely "implementation detail" that the consumers of those methods can be blissfully unaware of.

`WeakBench` itself is an `Activity`, invoking both Dalvik and native implementations of these two benchmarks. It uses a series of `AsyncTask` objects for executing the benchmarks on background threads, then updates `TextView` widgets in the UI to show the results:

```
package com.commonsware.android.tuning.weakbench;

import android.app.Activity;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.SystemClock;
import android.widget.TextView;

public class WeakBench extends Activity {
  static {
    System.loadLibrary("weakbench");
  }

  public native void nsievenative();
  public native void specnative();

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    new JavaSieveTask().execute();
  }

  /*
  * Code after this point is adapted from the Great Computer Language
  * Shootout. Copyrights are owned by whoever contributed this stuff,
  * or possibly the Shootout itself, since there isn't much information
  * on ownership there. Licensed under a modified BSD license.
  */
```

```
private class JavaSieveTask extends AsyncTask<Void, Void, Void> {
  long start=0;
  TextView result=null;

  @Override
  protected void onPreExecute() {
    result=(TextView)findViewById(R.id.nsieve_java);

    result.setText("running...");
  }

  @Override
  protected Void doInBackground(Void... unused) {
    start=SystemClock.uptimeMillis();

    int n=9;
    int m=(1<<n)*10000;
    boolean[] flags=new boolean[m+1];

    nsieve(m,flags);

    m=(1<<n-1)*10000;
    nsieve(m,flags);

    m=(1<<n-2)*10000;
    nsieve(m,flags);

    return(null);
  }

  @Override
  protected void onPostExecute(Void unused) {
    long delta=SystemClock.uptimeMillis()-start;

    result.setText(String.valueOf(delta));
    new JavaSpecTask().execute();
  }
}

private class JavaSpecTask extends AsyncTask<Void, Void, Void> {
  long start=0;
  TextView result=null;

  @Override
  protected void onPreExecute() {
    result=(TextView)findViewById(R.id.spec_java);

    result.setText("running...");
  }

  @Override
  protected Void doInBackground(Void... unused) {
    start=SystemClock.uptimeMillis();
```

```
      Approximate(1000);

      return(null);
    }

    @Override
    protected void onPostExecute(Void unused) {
      long delta=SystemClock.uptimeMillis()-start;

      result.setText(String.valueOf(delta));
      new JNISieveTask().execute();
    }
  }

  private class JNISieveTask extends AsyncTask<Void, Void, Void> {
    long start=0;
    TextView result=null;

    @Override
    protected void onPreExecute() {
      result=(TextView)findViewById(R.id.nsieve_jni);

      result.setText("running...");
    }

    @Override
    protected Void doInBackground(Void... unused) {
      start=SystemClock.uptimeMillis();

      nsievenative();

      return(null);
    }

    @Override
    protected void onPostExecute(Void unused) {
      long delta=SystemClock.uptimeMillis()-start;

      result.setText(String.valueOf(delta));
      new JNISpecTask().execute();
    }
  }

  private class JNISpecTask extends AsyncTask<Void, Void, Void> {
    long start=0;
    TextView result=null;

    @Override
    protected void onPreExecute() {
      result=(TextView)findViewById(R.id.spec_jni);

      result.setText("running...");
    }
    @Override
    protected Void doInBackground(Void... unused) {
      start=SystemClock.uptimeMillis();

      specnative();

      return(null);
    }

    @Override
    protected void onPostExecute(Void unused) {
      long delta=SystemClock.uptimeMillis()-start;

      result.setText(String.valueOf(delta));
    }
  }

  private static int nsieve(int m, boolean[] isPrime) {
      for (int i=2; i <= m; i++) isPrime[i] = true;
      int count = 0;

      for (int i=2; i <= m; i++) {
        if (isPrime[i]) {
            for (int k=i+i; k <= m; k+=i) isPrime[k] = false;
            count++;
        }
      }
      return count;
  }

  private final double Approximate(int n) {
    // create unit vector
    double[] u = new double[n];
    for (int i=0; i<n; i++) u[i] =  1;

    // 20 steps of the power method
    double[] v = new double[n];
    for (int i=0; i<n; i++) v[i] = 0;

    for (int i=0; i<10; i++) {
      MultiplyAtAv(n,u,v);
      MultiplyAtAv(n,v,u);
    }

    // B=AtA         A multiplied by A transposed
    // v.Bv /(v.v)   eigenvalue of v
    double vBv = 0, vv = 0;
    for (int i=0; i<n; i++) {
      vBv += u[i]*v[i];
      vv  += v[i]*v[i];
    }

    return Math.sqrt(vBv/vv);
  }


  /* return element i,j of infinite matrix A */
  private final double A(int i, int j){
    return 1.0/((i+j)*(i+j+1)/2 +i+1);
  }

  /* multiply vector v by matrix A */
  private final void MultiplyAv(int n, double[] v, double[] Av){
    for (int i=0; i<n; i++){
      Av[i] = 0;
      for (int j=0; j<n; j++) Av[i] += A(i,j)*v[j];
    }
  }

  /* multiply vector v by matrix A transposed */
  private final void MultiplyAtv(int n, double[] v, double[] Atv){
    for (int i=0;i<n;i++){
      Atv[i] = 0;
      for (int j=0; j<n; j++) Atv[i] += A(j,i)*v[j];
    }
  }

  /* multiply vector v by matrix A and then by matrix A transposed */
  private final void MultiplyAtAv(int n, double[] v, double[] AtAv){
    double[] u = new double[n];
    MultiplyAv(n,v,u);
    MultiplyAtv(n,u,AtAv);
  }
}
```

As with our C implementations of the benchmarks, the Java source code is derived from the Great Language Benchmarks Game.

Building and Deploying Your Project

Given that you have done all of this, the rest is perfectly normal – you build and deploy your Android project no differently than if you did not have any C/C++ code. Your native library is embedded in your APK file, so you do not have to worry about distributing it separately.

Bear in mind that your application will only work on platforms for which you have versions of your library. Since the vast majority of Android devices run ARM chipsets, this will not be a significant issue right now. As we encounter Atom-powered Google TVs and the like, in time, you may wish to adjust your `Application.mk` file to add in more platforms, so your application will be available for these non-ARM environments.

CHAPTER 4

Improving CPU Performance in Java

Knowing that you have CPU-related issues in your app is one thing – doing something about it is the next challenge. In some respects, tuning an Android application is a "one-off" job, tied to the particulars of the application and what it is trying to accomplish. That being said, this chapter will outline some general-purpose ways of boosting performance that may counter issues that you are running into.

Reduce CPU Utilization

One class of CPU-related problems come from purely sluggish code. These are the sorts of things you will see in Traceview, for example – methods or branches of code that seem to take an inordinately long time. These are also some of the most difficult to have general solutions for, as often times it comes down to what the application is trying to accomplish. However, the following sections provide suggestions for consuming fewer CPU instructions while getting the same work done.

These are presented in no particular order.

Standard Java Optimizations

Most of your algorithm fixes will be standard Java optimizations, no different than have been used by Java projects over the past decade and change. This section outlines a few of them. For more, consider reading *Effective Java* by Joshua Bloch or *Java Performance Tuning* by Jack Shirazi.

Avoid Excessive Synchronization

Few objects in `java.*` namespaces are intrinsically thread-safe, outside of `java.util.concurrent`. Typically, you need to perform your own synchronization if multiple threads will be accessing non-thread-safe objects. However, sometimes, Java classes have synchronization that you neither expect nor need. Synchronization adds unnecessary overhead.

The classic example here is `StringBuffer` and `StringBuilder`. `StringBuffer` was part of Java from early on, and, for whatever reason, was written to be thread-safe – two threads that append to the buffer will not cause any problems. However, most of the time, you are only using the `StringBuffer` from one thread, meaning all that synchronization overhead is a waste. Later on, Java added `StringBuilder`, with the same basic set of methods as has `StringBuffer`, but without the synchronization.

Similarly, in your own code, only synchronize where is is really needed. Do not toss the `synchronized` keyword around randomly, or use concurrent collections that will only be used by one thread, etc.

Avoid Floating-Point Math

The first generation of Android devices lacked a floating-point coprocessor on the ARM CPU package. As a result, floating-point math speed was atrocious. That is why the Google Maps add-on for Android uses `GeoPoint`, with latitude and longitude in integer microdegrees, rather than the standard Android `Location` class, which uses Java double variables holding decimal degrees.

While later Android devices do have floating-point coprocessor support, that does not mean that floating-point math is now as fast as integer math. If you find that your code is spending lots of time on floating-point calculations, consider whether a change in units would allow you to replace the floating-point calculations with integer equivalents. For example, microdegrees for latitude and longitude provide adequate granularity for most maps, yet allow Google Maps to do all of its calculations in integers.

Similarly, consider whether the full decimal accuracy of floating-point values is really needed. While it may be physically possible to perform distance calculations in meters with accuracy to a few decimal points, for example, in many cases the user will not need that degree of accuracy. If so, perhaps changing to fixed-point (integer) math can boost your performance.

Don't Assume Built-In Algorithms are Best

Years upon years of work has gone into the implementation of various algorithms that underlie Java methods, like searching for substrings inside of strings.

Somewhat less work has gone into the implementation of the Apache Harmony versions of those methods, simply because the project is younger, and it is a modified version of the Harmony implementation that you will find in Android. While the core Android team has made many improvements to the original Harmony implementation, those improvements may be for optimizations that do not fit your needs (e.g., optimizing to reduce memory consumption at the expense of CPU time).

But beyond that, there are dozens of string-matching algorithms, some of which may be better for you depending on the string being searched and the string being searched for. Hence, you may wish to consider applying your own searching algorithm rather than relying on the built-in one, to boost performance. And, this same concept may hold for other algorithms as well (e.g., sorting).

Of course, this will also increase the complexity of your application, with long-term impacts in terms of maintenance cost. Hence, do not assume the built-in algorithms are the worst, either – optimize those algorithms that Traceview or logging suggest are where you are spending too much time.

Support Hardware-Accelerated Graphics

An easy "win" is to add `android:hardwareAccelerated="true"` to your `<application>` element in the manifest. This toggles on hardware acceleration for 2D graphics, including much of the stock widget framework. For maximum backwards compatibility, this hardware acceleration is off, but adding the aforementioned attribute will enable it for all activities in your application.

Note that this is only available starting with Android 3.0. It is safe to have the attribute in the manifest for older Android devices, as they simply will ignore your request.

You also should test your application thoroughly after enabling hardware acceleration, to make sure there are no unexpected issues. For ordinary widget-based applications, you should encounter no problems. Games or other applications that do their own drawing might have issues. If you find that some of your code runs into problems, you can override hardware acceleration on a per-activity basis by putting the `android:hardwareAccelerated` on `<activity>` elements in the manifest.

Minimize IPC

Calling a method on an object in your own process is fairly inexpensive. The overhead of the method invocation is fairly minuscule, and so the time involved is simply however long it takes for that method to do its work.

Invoking behaviors in another process, via inter-process communication (IPC), is considerably more expensive. Your request has to be converted into a byte array (e.g., via the `Parcelable` interface), made available to the other process, converted back into a regular request, then executed. This adds substantial CPU overhead.

There are three basic flavors of IPC in Android:

- "Directly" invoking a third-party application's service's AIDL-published interface, to which you bound with `bindService()`
- Performing operations on a content provider that is not part of your application (i.e., supplied by the OS or a third-party application)
- Performing other operations that, under the covers, trigger IPC

Remote Bound Service

Using a remote service is fairly obvious when you do it – it is difficult to mistake copying the AIDL into your project and such. The proxy object generated from the AIDL converts all your method calls on the interface into IPC operations, and this is relatively expensive.

If you are exposing a service via AIDL, design your API to be coarse-grained. Do not require the client to make 1,000 method invocations to accomplish something that can be done in 1 via slightly more complex arguments and return values.

If you are consuming a remote service, try not to get into situations where you have to make lots of calls in a tight loop, or per row of a scrolled `AdapterView`, or anything else where the overhead may become troublesome.

For example, in the `CPU-Java/AIDLOverhead` directory of the book's source code, you will find a pair of projects implementing the same do-nothing method in equivalent services. One uses AIDL and is bound to remotely from a separate client application; the other is a local service in the client application itself. The client then calls the do-nothing method 1 million times for each of the two services. On average, on a Samsung Galaxy Tab 10.1, 1 million calls takes around 170 seconds for the remote service, while it takes around 170 *milliseconds* for the local service. Hence, the overhead of an individual remote method invocation is small (~170 microseconds), but doing lots of them in a loop, or as the user flings a `ListView`, might become noticeable.

Remote Content Provider

Using a content provider can be somewhat less obvious of a problem. Using `ContentResolver` or `managedQuery()` or a `CursorLoader` looks the same whether it is your own content provider or someone else's. However, you know what content providers you wrote; anything else is probably running in another process.

As with remote services, try to aggregate operations with remote content providers, such as:

- Use `bulkInsert()` rather than lots of individual `insert()` calls
- Try to avoid calling `update()` or `delete()` in a tight loop – instead, if the content provider supports it, use a more complex "WHERE clause" to update or delete everything at once
- Try to get all your data back in few queries, rather than lots of little ones… though this can then cause you issues in terms of memory consumption

Remote OS Operation

The content provider scenario is really a subset of the broader case where you request that Android do something for you and winds up performing IPC as part of that.

Sometimes, this is going to be obvious. If you are sending commands to a third-party service via `startService()`, by definition, this will involve IPC, since the third-party service will run in a third-party process. Try to avoid calling `startService()` lots of times in close succession.

However, there are plenty of cases that are less obvious:

- All requests to `startActivity()`, `startService()`, and `sendBroadcast()` involve IPC, as it is a separate OS process that does the real work
- Registering and unregistering a `BroadcastReceiver` (e.g., `registerReceiver()`) involves IPC
- All of the "system services", such as `LocationManager`, are really rich interfaces to an AIDL-defined remote service, and so most operations on these system services require IPC

Once again, your objective should be to minimize calls that involve IPC, particularly where you are making those calls frequently in close succession, such as in a loop. For example, frequently calling `getLastKnownLocation()` will be expensive, as that involves IPC to a system process.

Android-Specific Java Optimizations

The way that the Dalvik VM was implemented and operates is subtly different than a traditional Java VM. Therefore, there are some optimizations that are more important on Android than you might find in regular desktop or server Java.

The Android developer documentation has a roster of such optimizations. Some of the highlights include:

- Getters and setters, while perhaps useful for encapsulation, are significantly slower than direct field access. For simpler cases, such as `ViewHolder` objects for optimizing an Adapter, consider skipping the accessor methods and just use the fields directly.
- Some popular method calls are replaced by hand-created assembler instructions rather than code generated via the JIT compiler. `indexOf()` on `String` and `arraycopy()` on `System` are two cited examples. These will run much faster than anything you might create yourself in Java.

Reduce Time on the Main Application Thread

Another class of CPU-related problem is when your code may be efficient, but it is occurring on the main application thread, causing your UI to react sluggishly. You might have tuned your decryption algorithm as best as is mathematically possible, but it may be that decrypting data on the main application thread simply takes too much time. Or, perhaps `StrictMode` complained about some disk or network I/O that you are performing on the main application thread.

The following sections recap some commonly-seen patterns for moving work off the main application thread, plus a few newer options that you may have missed.

Generate Less Garbage

Most developers think of having too many allocations as being solely an issue of heap space. That certainly has an impact, and depending on the nature of the allocations (e.g., bitmaps), it may be the dominant issue.

However, garbage has impacts from a CPU standpoint as well. Every object you create causes its constructor to be executed. Every object that is garbage-collected requires CPU time both to find the object in the heap and to actually clean it up (e.g., execute the finalizer, if any).

Worse still, on older versions of Android (e.g., Android 2.2 and down), the garbage collector interrupts the entire process to do its work, so the more garbage you generate, the more times you "stop the world". Game developers have had to deal with this since Android's inception. To maintain a 60 FPS refresh rate, you cannot afford *any* garbage collections on older devices, as a single GC run could easily take more than the ~16ms you have per drawing pass.

As a result of all of this, game developers have had to carefully manage their own object pools, pre-allocating a bunch of objects before game play begins, then using and recycling those objects themselves, only allowing them to become garbage after game play ends.

Most non-game Android applications may not have to go to quite that extreme across the board. However, there are cases where excessive allocation may cause you difficulty. For example, avoiding creating too much garbage is one aspect of view recycling with `AdapterView`, which is covered in greater detail in the next section.

If Traceview indicates that you are spending a lot of time in garbage collection, pay attention to your loops or things that may be invoked many times in rapid succession (e.g., accessing data from a custom `Cursor` implementation that is tied to a `CursorAdapter`). These are the most likely places where your own code might be creating lots of extra objects that are not needed. Examining the heap to see what is all being created (and eventually garbage collected) will be covered in an upcoming chapter of the book.

View Recycling

Perhaps the best-covered Android-specific optimization is view recycling with `AdapterView`. This is discussed in any decent introductory Android text, including the author's own The Busy Coder's Guide to Android Development.

In a nutshell, if you are extending `BaseAdapter`, or if you are overriding `getView()` in another adapter, please make use of the `View` parameter supplied to `getView()` (referred to here as `convertView`). If `convertView` is not `null`, it is one of your previous `View` objects you returned from `getView()` before, being offered to you for recycling purposes. Using `convertView` saves you from inflating or manually constructing a fresh `View` every time the user scrolls, and both of those operations are relatively expensive.

If you have been ignoring `convertView` because you have more than one type of `View` that `getView()` returns, your `Adapter` should be overriding `getViewTypeCount()` and `getItemViewType()`. These will allow Android to maintain separate object pools for each type of row from your `Adapter`, so `getView()` is guaranteed to be passed a `convertView` that matches the row type you are trying to create.

A somewhat more advanced optimization – caching all those `findViewById()` lookups – is also possible once your row recycling is in place. Often referred to as "the holder pattern", you do the `findViewById()` calls when you inflate a new row, then attach the `findViewById()` results to the row itself via some custom "holder" object and the `setTag()` method on `View`. When you recycle the row, you can get your "holder" back via `getTag()` and skip having to do the `findViewById()` calls again.

Not only do these techniques save CPU time overall, but all of this work is always done on the main application thread, so it significantly boosts perceived performance.

Background Threads

Of course, the backbone of any strategy to move work off the main application thread is to use background threads, in one form or fashion. You will want to apply these in places where `StrictMode` complains about network or disk I/O, or places where Traceview or logging indicate that you are taking too much time on the main application thread during GUI processing (e.g., converting downloaded bitmap images into `Bitmap` objects via `BitmapFactory`).

Sometimes, you will manually dictate where work should be done in the background, either by forking threads yourself or by using `AsyncTask`. `AsyncTask` is a nice framework, handling all of the inter-thread communication for you and neatly packaging up the work to be done in readily understood methods. However, `AsyncTask` does not fit every scenario – it is mostly designed for "transactional" work that is known to take a modest amount of time (milliseconds to seconds) then end. For cases where you need unbounded background processing, such as monitoring a socket for incoming data, forking your own thread will be the better approach.

Sometimes, you will use facilities supplied by Android to move work to the background. For example, many activities are backed by a `Cursor` obtained from a database or content provider. Classically, you would manage the cursor (via `startManagingCursor()`) or otherwise arrange to refresh that `Cursor` in `onResume()`, so when your activity returns to the foreground after having been gone for a while, you would have fresh data. However, this pattern tends to lead to database I/O on the main application thread, triggering complaints from `StrictMode`. Android 3.0 and the Android Compatibility Library offer a `Loader` framework designed to try to solve the core pattern of refreshing the data, while arranging for the work to be done asynchronously.

<< TBD: Loader sample >>

Asynchronous BroadcastReceiver Operations

99.44% of the time (approximately) that Android calls your code in some sort of event handler, you are being called on the main application thread. This includes manifest-registered `BroadcastReceiver` components – `onReceive()` is called on the main application thread. So any work you do in `onReceive()` ties up that thread (possibly impacting an activity of yours in the foreground), and if you take more than 10 seconds, Android will terminate your `BroadcastReceiver` with extreme prejudice.

Classically, manifest-registered `BroadcastReceiver` components only live as long as the `onReceive()` call does, meaning you can do very little work in the `BroadcastReceiver` itself. The typical pattern is to have it send a command to a service via `startService()`, where the service "does the heavy lifting".

Android 3.0 added a `goAsync()` method on `BroadcastReceiver` that can help a bit here. While under-documented, it tells Android that you need more time to complete the broadcast work, but that you can do that work on a background thread. This does not eliminate the 10-second rule, but it does mean that the `BroadcastReceiver` can do some amount of I/O without having to send a command to a service to do it while still not tying up the main application thread.

The CPU-Java/GoAsync sample project in the book's source code demonstrates goAsync() in use, as the project name might suggest.

Our activity's layout consists of two Button widgets and an EditText widget:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <EditText android:id="@+id/editText1" android:layout_width="match_parent"
    android:layout_height="wrap_content">
  </EditText>
  <Button android:layout_width="match_parent" android:id="@+id/button1"
    android:layout_height="wrap_content" android:text="@string/nonasync"
    android:onClick="sendNonAsync"></Button>
  <Button android:layout_width="match_parent" android:id="@+id/button2"
    android:layout_height="wrap_content" android:text="@string/async"
    android:onClick="sendAsync"></Button>
</LinearLayout>
```

The activity itself simply has sendAsync() and sendNonAsync() methods, each invoking sendBroadcast() to a different BroadcastReceiver implementation:

```
package com.commonsware.android.tuning.goasync;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class GoAsyncActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void sendAsync(View v) {
    sendBroadcast(new Intent(this, AsyncReceiver.class));
  }

  public void sendNonAsync(View v) {
    sendBroadcast(new Intent(this, NonAsyncReceiver.class));
  }
}
```

The NonAsyncReceiver simulates doing time-consuming work in onReceive() itself:

```
package com.commonsware.android.tuning.goasync;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class NonAsyncReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context arg0, Intent arg1) {
    SystemClock.sleep(7000);
  }
}
```

Hence, if you click the "Send Non-Async Broadcast" button, not only will the button fail to return to its normal state for seven seconds, but the `EditText` will not respond to user input either.

The `AsyncReceiver`, though, uses `goAsync()`:

```
package com.commonsware.android.tuning.goasync;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class AsyncReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    final BroadcastReceiver.PendingResult result=goAsync();

    (new Thread() {
      public void run() {
        SystemClock.sleep(7000);
        result.finish();
      }
    }).start();
  }
}
```

The `goAsync()` method returns a `PendingResult`, which supports a series of methods that you might ordinarily fire on the `BroadcastReceiver` itself (e.g., `abortBroadcast()`) but want to do on a background thread. You need your background thread to have access to the `PendingResult` – in this case, via a `final` local variable. When you are done with your work, call `finish()` on the `PendingResult`.

If you click the "Send Async Broadcast" button, even though we are still sleeping for 7 seconds, we are doing so on a background thread, and so our user interface is still responsive.

Saving SharedPreferences

The classic way to save `SharedPreferences.Editor` changes was via a call to `commit()`. This writes the preference information to an XML file on whatever thread you are on – another hidden source of disk I/O you might be doing on the main application thread.

If you are on API Level 9, and you are willing to blindly try saving the changes, use the new `apply()` method on `SharedPreferences.Editor`, which works asynchronously.

If you need to support older versions of Android, or you really want the boolean return value from `commit()`, consider doing the `commit()` call in an `AsyncTask` or background thread.

And, of course, to support both of these, you will need to employ tricks like conditional class loading. You can see that used for saving `SharedPreferences` in the `CPU-Java/PrefsPersist` project in the book's source code. The activity reads in a preference, puts the current value on the screen, then updates the preference with the help of an `AbstractPrefsPersistStrategy` class and its `persist()` method:

```
package com.commonsware.android.tuning.prefs;

import android.app.Activity;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.widget.TextView;

public class PrefsPersistActivity extends Activity {
  private static final String KEY="counter";

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    SharedPreferences prefs=
      PreferenceManager.getDefaultSharedPreferences(this);
    int counter=prefs.getInt(KEY, 0);

    ((TextView)findViewById(R.id.value)).setText(String.valueOf(counter));

    AbstractPrefsPersistStrategy.persist(prefs.edit().putInt(KEY, counter+1));
  }
}
```

AbstractPrefsPersistStrategy is an abstract base class that will hold a strategy implementation, depending on Android version. On pre-Honeycomb builds, it uses an implementation that forks a background thread to perform the commit():

```
package com.commonsware.android.tuning.prefs;

import android.content.SharedPreferences;
import android.os.Build;

abstract public class AbstractPrefsPersistStrategy {
  abstract void persistAsync(SharedPreferences.Editor editor);

  private static final AbstractPrefsPersistStrategy INSTANCE=initImpl();

  public static void persist(SharedPreferences.Editor editor) {
    INSTANCE.persistAsync(editor);
  }

  private static AbstractPrefsPersistStrategy initImpl() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk<Build.VERSION_CODES.HONEYCOMB) {
      return(new CommitAsyncStrategy());
    }

    return(new ApplyStrategy());
  }

  static class CommitAsyncStrategy extends AbstractPrefsPersistStrategy {
    @Override
    void persistAsync(final SharedPreferences.Editor editor) {
      (new Thread() {
        @Override
        public void run() {
          editor.commit();
        }
      }).start();
    }
  }
}
```

On Honeycomb and higher, it uses a separate strategy class that uses the new `apply()` method:

```
package com.commonsware.android.tuning.prefs;

import android.content.SharedPreferences.Editor;

public class ApplyStrategy extends AbstractPrefsPersistStrategy {

  @Override
  void persistAsync(Editor editor) {
    editor.apply();
  }
}
```

By separating the Honeycomb-specific code out into a separate class, we can avoid loading it on older devices and encountering the dreaded `VerifyError`.

Whether using the built-in `apply()` method is worth dealing with multiple strategies, versus simply calling `commit()` on a background thread, is up to you.

Improve Throughput and Responsiveness

Being efficient and doing work on the proper thread may still not be enough. It could be that your work is not consuming excessive CPU time, but is taking too long in "wall clock time" (e.g., the user sits waiting too long at a `ProgressDialog`). Or, it could be that your work, while efficient and in the background, is causing difficulty for foreground operations.

The following sections outline some common problems and solutions in this area.

Minimize Disk Writes

Earlier in this book, we emphasized moving disk writes off to background threads.

Even better is to get rid of some of the disk writes entirely.

A big culprit here comes in the form of database operations. By default, each `insert()`, `update()`, or `delete()`, or any `execSQL()` invocation that modifies data, will occur in its own transaction. Each transaction involves a set of disk writes. Many times, this is not a problem. But, if you are doing a lot of these – such as importing records from a CSV file – hundreds or thousands of transactions will mean thousands of individual disk writes, and that can take some time. You may wish to wrap those operations in your own transaction, using methods like `beginTransaction()`, simply to reduce the number of transactions and, therefore, disk writes.

If you are doing your own disk I/O beyond databases, you may encounter similar sorts of issues. Overall, it is better to do a few larger writes than lots of little ones.

Set Thread Priority

Threads you fork, by default, run at a default priority: `THREAD_PRIORITY_DEFAULT` as defined on the `Process` class. This is a lower priority than the main application thread (`THREAD_PRIORITY_DISPLAY`).

Threads you use via `AsyncTask` run at a lower priority (`THREAD_PRIORITY_BACKGROUND`). If you fork your own threads, then, you might wish to consider moving them to a lower priority as well, to affect how much time they get compared to the main application thread. You can do this via `setThreadPriority()` on the `Process` class.

The lowest possible priority, `THREAD_PRIORITY_LOWEST`, is described as "only for those who really, really don't want to run if anything else is happening".

You might use this for "idle-time processing", but bear in mind that the thread will be paused a lot to allow other threads to run.

Lower-priority threads will help ensure that your background work does not affect your foreground UI. Processes themselves are put in a lower-priority class as they move to the background (e.g., you have no activities visible), which further reduces the amount of CPU time you will be using at any given moment.

Also, note that `IntentService` uses a thread at default (*not* background) priority – you may wish to drop the priority of this thread to something that will be lower than your main application thread, to minimize how much CPU time the `IntentService` steals from your UI.

Do the Work Some Other Time

Just because you could do the work now does not mean you should do the work now. Perhaps a better answer is to do the work later, or do part of the work now and part of the work later.

For example, suppose that you have your own database of points of interest for your custom map application. Periodically, you publish a new database on your Web site, which your Android app should download. Odds are decent that the user is not in desperate need for this new database right away. In fact, the CPU time and disk I/O time to download and save the database might incrementally interfere with the foreground application, despite your best efforts.

In this case, not only should you check for and download the database when the user is unlikely to be using the device (e.g., before dawn), but you should check whether the screen is on via `isScreenOn()` on `PowerManager`, and delay the work to sometime when the screen is off. For example, you could have `AlarmManager` set up to have your code check for updates every 24 hours at 4am. If, at 4am, the screen is on, your code could skip the download and wait until tomorrow, or skip the download and add a oneshot alarm to wake you up in 30 minutes, in hopes that the user will no longer be using the device.

At the same time, you may wish to consider having a "refresh" menu choice somewhere, for when the user specifically wants you to go get the update (if available) *now*, for whatever reason.

PART II – Bandwidth Performance

CHAPTER 5

Issues with Bandwidth

As anyone who owned an Apple Newton or Palm V PDA back in the 1990's, handheld devices have been around for quite some time. For a very long time, they were a niche product, associated with geeks, nerds, and the occasional business executive.

Internet access changed all of that.

Blackberry for enterprise messaging – an outgrowth of its original two-way paging approach – blazed part of the trail, but the concept "crossed the chasm" to ordinary people with the advent of the iPhone, Android devices, and similar equipment.

Therefore, it is not terribly surprising when Android developers want to add Internet capabilities to their apps. To the contrary, it is almost unusual when you encounter an app that does *not* want to use the Internet for something or another.

However, mobile Internet access inherits all of the classic problems of Internet access (e.g., "server not found") and adds new and exciting challenges, all of which can leave a developer with an app that has performance issues.

You're Using Too Much of the Slow Stuff

To paraphrase America's Founding Fathers, "all Internet connections are not created equal".

One form of inequality is speed. Different classes of connection have different theoretical upper bounds. WiMAX and other "4G" connections are theoretically faster than 3G connections, which are theoretically faster than 2G or EDGE connections. WiFi – typically 802.11g in today's devices – is theoretically ridiculously fast though it is typically limited by the ISP connection, and ISP connections can run the gamut from really fast to merely good.

However, "theoretical" bounds tend to run afoul of reality. There are plenty of places where high-speed mobile data connections are non-existent, despite what the carriers' coverage maps claim. 2G mobile data works, but is not especially speedy. This layers on top of the typical Internet congestion issues, along with typically transitory problems (e.g., trying to get connectivity while attending a technology conference keynote presentation).

Hence, what runs quickly in the lab may run much more slowly in users' hands.

If you followed the instructions in previous chapters on CPU bottlenecks, the limited bandwidth will not cause your UI to become "janky", in that it will be responsive to touches and taps. However, poor connectivity will mean that you are simply slow to respond to user requests. For example, clicking the "check for new email" menu button has no immediate effect. If you feel that you need a splash screen or progress indicator to tell the user that "we are really checking for new email, honest", then you know that your Internet access is slower than is ideal.

Obviously, some of this is unavoidable. However, the objective of the chapters in this part of the book is to give you an idea of ways to reduce your bandwidth consumption, making those delays be that much less annoying for your users.

You're Using Too Much of the Expensive Stuff

Mobile data tends to come with more strings attached than does WiFi.

In the US, it used to be that mobile data connections included unlimited usage. Now, at best, a mobile data plan has "unlimited" usage for a curious definition of the term "unlimited". More and more carriers are moving towards a hard cap – go above the cap, and you either cannot use more bandwidth, have your speeds curtailed, or pay significantly for additional bandwidth.

Outside of the US, the "pay significantly for bandwidth" approach is fairly typical. So-called "metered" data plans simply charge you such-and-so per MB or GB of bandwidth.

And, to top it off, roaming almost always is a metered plan. So, a US resident traveling overseas, even with a SIM and phone that supports international usage, would pay a ridiculous sum for bandwidth. Stories of phone bills in the tens of thousands of dollars abound, where people simply used their phone as they normally would when they were outside of their home network.

Hence, if you use a fair bit of bandwidth, it would be really nice if you offered users means to consume less of it when they are on mobile data compared to WiFi (which is typically unmetered). You could elect to poll your server less frequently, for example, giving the users the ability to specify separate polling periods depending on which type of connection they have.

And, of course, there are other "costs" for using bandwidth besides direct monetary costs. For example, downloading data over a slower mobile data connection may consume more power than downloading the same data over WiFi – while the WiFi radio might consume additional power, the time difference might account for more power consumption, if the CPU could be powered down for the rest of that time.

These chapters will show you how you can react to changes in connectivity and approaches for how to use that information to reduce costs for the user.

You're Using Too Much of Somebody Else's Stuff

It is easy for developers to think that they alone are using a user's device. Alas, this is infrequently the case, particularly when it comes to background Internet access.

While your application is busily downloading stuff, some other application might be busily downloading stuff. In principle, this should not be an issue, as multiple applications can access the Internet simultaneously. However, bandwidth can become an issue. If you are in the background, and the other application is in the foreground, the user might notice that bandwidth is an issue. For example, users might be unhappy if your downloads are impeding their ability to watch streaming video, or play their favorite Android-based MMORPG, or whatever.

A polite Android application will test to see whether the foreground application is heavily using the Interent and will curtail its own Internet use while that is going on. This chapter will help you learn how to make that determination and how to respond.

You're Using Too Much... And There Is None

Not only might location dictate how much bandwidth you have, but whether you have any bandwidth at all.

While some people think that the entire planet has connectivity, reality once again dictates otherwise. Major metropolitan areas have connectivity… at least, so long as the carriers have not melted down due to overuse, as AT&T tended to do during the early months of the iPhone Invasion. Outlying areas are much more hit-or-miss. Voice is sometimes a challenge, let alone data. And it only *seems* as though there is a Starbucks every 100 meters, which might actually provide blanket WiFi coverage.

Then, of course, there are planes (most do not offer in-flight WiFi at this time), international travel without an international-capable phone plan, and so on.

Some Android applications have the potential to still offer near-complete functionality despite this, with a bit of user assistance. For example, Google Maps for Android now has an offline caching feature, which will download data for a 10-mile radius from a given point, for use while the device is otherwise offline.

Here, the issue becomes less one of bandwidth (other than detecting that you have no connection) and more one of caching and storage. The space-related issues that these techniques can raise will be covered elsewhere in this book.

CHAPTER 6

Focus On: TrafficStats

To be able to have more intelligent code – code that can adapt to Internet activity on the device – Android offers the `TrafficStats` class. This class really is a gateway to a block of native code that reports on traffic usage for the entire device and per-application, for both received and transmitted data. This chapter will examine how you can access `TrafficStats` and interpret its data.

TrafficStats Basics

The `TrafficStats` class is not designed to be instantiated – you will not be invoking a constructor by calling `new TrafficStats()` or something like that. Rather, `TrafficStats` is merely a collection of static methods, mapped to native code, that provide access to point-in-time traffic values. No special permissions are needed to use any of these methods. Most of the methods were added in API Level 8 and therefore should be callable on most Android devices in use today.

Device Statistics

If you are interested in overall traffic, you will probably care most about the `getTotalRxBytes()` and `getTotalTxBytes()` on `TrafficStats`. These methods return received and transmitted traffic, respectively, measured in bytes.

You also have:

- `getTotalRxPackets()` and `getTotalTxPackets()`, if for your case measuring IP packets is a better measure than bytes
- `getMobileRxBytes()` and `getMobileTxBytes()`, which return the traffic going over mobile data (also included in the total)
- `getMobileRxPackets()` and `getMobileTxPackets()`, which are the packet counts for the mobile data connection

Per-Application Statistics

Technically, `TrafficStats` does not provide per-application traffic statistics. Rather, it provides per-UID traffic statistics. In most cases, the UID (user ID) of an application is unique, and therefore per-UID statistics map to per-application statistics. However, it is possible for multiple applications to share a single UID (e.g., via the `android:sharedUserId` manifest attribute) – in this case, `TrafficStats` would appear to provide traffic data for all applications sharing that UID.

There are per-UID equivalents of the first four methods listed in the previous section, replacing "Total" with "Uid". So, to find out overall traffic for an application, you could use `getUidRxBytes()` and `getUidTxBytes()`. However, these are the only two UID-specific methods that were implemented in API Level 8. Equivalents of the others (e.g., `getUidRxPackets()`) were added in API Level 12. API Level 12 also added some TCP-specific methods (e.g., `getUidTcpTxBytes()`). Note, though, that the mobile-only method are only available at the device level; there are no UID-specific versions of those methods.

Interpreting the Results

You will get one of two types of return value from these methods.

In theory, you will get the value the method calls for (e.g., number of bytes, number of packets). The documentation does not state the time period for that value, so while it is possible that it is really "number of bytes since the device was booted", we do not know that for certain. Hence, `TrafficStats` results should be used for comparison purposes, either comparing the same value over time or comparing multiple values at the same time. For example, to measure bandwidth consumption, you will need to record the `TrafficStats` values at one point in time, then again later – the difference between them represents the consumed bandwidth during that period of time.

In practice, while the "total" methods seem reliable, the per-UID methods often return `-1`. The official explanation for this is that the particular traffic metric is unavailable on that device, and this does explain some of the `-1` values that are returned. For example, a Nexus One running Android 2.3 returns `-1` for all the per-UID methods, while a Nexus S running Android 2.3 will return a positive value for *some* UIDs. It is unclear what the other `-1` values mean. Two possible meanings are:

- There has been no traffic of that type on that UID since boot, or
- You do not have permission to know the traffic of that type on that UID

Hence, the per-UID values are a bit "hit or miss", which you will need to take into account.

Example: TrafficMonitor

To illustrate the use of `TrafficStats` methods and analysis, let us walk through the code associated with the `TrafficMonitor` sample application. This is a simple activity that records a snapshot of the current traffic levels on startup, then again whenever you tap a button. On-screen, it will display the current value, previous value, and difference ("delta") between them. In LogCat, it will dump the same information on a per-UID basis.

TrafficRecord

It would have been nice if `TrafficStats` were indeed an object that you would instantiate, that captured the traffic values at that moment in time.

Alas, that is not how it was written, so we need to do that ourselves. In the `TrafficMonitor` project, this job is delegated to a `TrafficRecord` class:

```
package com.commonsware.android.tuning.traffic;

import android.net.TrafficStats;

class TrafficRecord {
  long tx=0;
  long rx=0;
  String tag=null;

  TrafficRecord() {
    tx=TrafficStats.getTotalTxBytes();
    rx=TrafficStats.getTotalRxBytes();
  }

  TrafficRecord(int uid, String tag) {
    tx=TrafficStats.getUidTxBytes(uid);
    rx=TrafficStats.getUidRxBytes(uid);
    this.tag=tag;
  }
}
```

There are two separate constructors, one for the total case and one for the per-UID case. The total case just logs `getTotalRxBytes()` and `getTotalTxBytes()`, while the per-UID case uses `getUidRxBytes()` and `getUidTxBytes()`. The per-UID case also stores a "tag", which is simply a `String` identifying the UID for this record – as you will see, `TrafficMonitor` uses this for a package name.

TrafficSnapshot

An individual `TrafficRecord`, though, is insufficient to completely capture the traffic figures at a moment in time. We need a collection of `TrafficRecord` objects, one for the device ("total") and one per running UID. The work to collect all of that is handled by a `TrafficSnapshot` class:

```
package com.commonsware.android.tuning.traffic;

import java.util.HashMap;
import android.content.Context;
import android.content.pm.ApplicationInfo;

class TrafficSnapshot {
  TrafficRecord device=null;
  HashMap<Integer, TrafficRecord> apps=
    new HashMap<Integer, TrafficRecord>();

  TrafficSnapshot(Context ctxt) {
    device=new TrafficRecord();

    HashMap<Integer, String> appNames=new HashMap<Integer, String>();

    for (ApplicationInfo app :
          ctxt.getPackageManager().getInstalledApplications(0)) {
      appNames.put(app.uid, app.packageName);
    }

    for (Integer uid : appNames.keySet()) {
      apps.put(uid, new TrafficRecord(uid, appNames.get(uid)));
    }
  }
}
```

The constructor uses `PackageManager` to iterate over all installed applications and builds up a `HashMap`, mapping the UID to a `TrafficRecord` for that UID, tagged with the application package name (e.g., `com.commonsware.android.tuning.traffic`). It also creates one `TrafficRecord` for the device as a whole.

TrafficMonitorActivity

`TrafficMonitorActivity` is what creates and uses `TrafficSnapshot` objects. This is a fairly conventional activity with a TableLayout-based UI:

```
<?xml version="1.0" encoding="utf-8"?>
  <TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/table" android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <Button android:text="Take Snapshot" android:onClick="takeSnapshot" />
    <TableRow>
      <TextView android:layout_column="1"
        android:text="@string/received" android:layout_gravity="right"
android:textSize="20sp"/>
      <TextView android:text="@string/sent" android:layout_gravity="right"
android:textSize="20sp"/>
    </TableRow>
    <TableRow>
      <TextView
        android:gravity="right" android:text="@string/latest"
        android:textStyle="bold"
android:layout_marginRight="@dimen/margin_right" android:textSize="20sp"/>
```

```
      <TextView android:id="@+id/latest_rx" android:gravity="right"
android:layout_marginRight="@dimen/margin_right" android:textSize="20sp"/>
      <TextView android:id="@+id/latest_tx" android:gravity="right"
android:textSize="20sp"/>
    </TableRow>
    <TableRow>
      <TextView
        android:gravity="right" android:text="@string/previous"
        android:textStyle="bold"
android:layout_marginRight="@dimen/margin_right" android:textSize="20sp"/>
      <TextView android:id="@+id/previous_rx" android:gravity="right"
android:layout_marginRight="@dimen/margin_right" android:textSize="20sp"/>
      <TextView android:id="@+id/previous_tx" android:gravity="right"
android:textSize="20sp"/>
    </TableRow>
    <TableRow>
      <TextView
        android:gravity="right" android:text="@string/delta"
        android:textStyle="bold"
android:layout_marginRight="@dimen/margin_right" android:textSize="20sp"/>
      <TextView android:id="@+id/delta_rx" android:gravity="right"
android:layout_marginRight="@dimen/margin_right" android:textSize="20sp"/>
      <TextView android:id="@+id/delta_tx" android:gravity="right"
android:textSize="20sp"/>
    </TableRow>
  </TableLayout>
```

The activity implementation consists of three methods. There is your typical `onCreate()` implementation, where we initialize the UI, get our hands on the `TextView` widgets for output, and take the initial snapshot:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  latest_rx=(TextView)findViewById(R.id.latest_rx);
  latest_tx=(TextView)findViewById(R.id.latest_tx);
  previous_rx=(TextView)findViewById(R.id.previous_rx);
  previous_tx=(TextView)findViewById(R.id.previous_tx);
  delta_rx=(TextView)findViewById(R.id.delta_rx);
  delta_tx=(TextView)findViewById(R.id.delta_tx);

  takeSnapshot(null);
}
```

The `takeSnapshot()` method creates a new `TrafficSnapshot` (held in a `latest` data member) after moving the last `TrafficSnapshot` to a `previous` data member. It then updates the `TextView` widgets for the `latest` data and, if the `previous` data member is not `null`, also for the `previous` snapshot and the difference between them. This alone is sufficient to update the UI, but we also want to log per-UID data to LogCat:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  latest_rx=(TextView)findViewById(R.id.latest_rx);
  latest_tx=(TextView)findViewById(R.id.latest_tx);
  previous_rx=(TextView)findViewById(R.id.previous_rx);
  previous_tx=(TextView)findViewById(R.id.previous_tx);
  delta_rx=(TextView)findViewById(R.id.delta_rx);
  delta_tx=(TextView)findViewById(R.id.delta_tx);

  takeSnapshot(null);
}
```

One possible problem with the snapshot system is that the process list may change between snapshots. One simple way to address this is to only log to LogCat data where the application's UID exists in both the `previous` and `latest` snapshots. Hence, `takeSnapshot()` uses a `HashSet` and `retainAll()` to determine which UIDs exist in both snapshots. For each of those, we call an `emitLog()` method to record the data to an `ArrayList`, which is then sorted and dumped to LogCat.

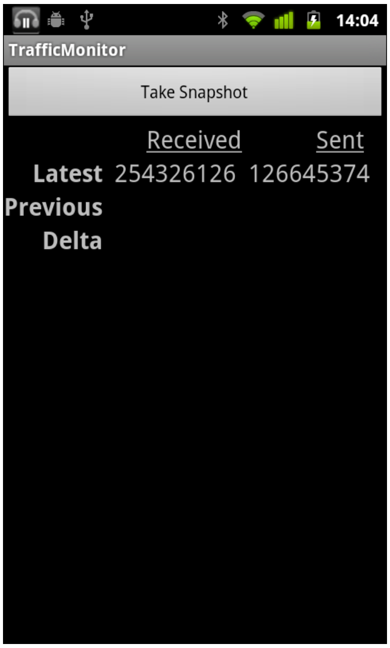
The `emitLog()` method builds up a line with the package name and bandwidth consumption information, assuming that there is bandwidth to report (i.e., we have a value other than `-1`):

```
private void emitLog(CharSequence name, TrafficRecord latest_rec,
                     TrafficRecord previous_rec,
                     ArrayList<String> rows) {
  if (latest_rec.rx>-1 || latest_rec.tx>-1) {
    StringBuilder buf=new StringBuilder(name);

    buf.append("=");
    buf.append(String.valueOf(latest_rec.rx));
    buf.append(" received");

    if (previous_rec!=null) {
      buf.append(" (delta=");
      buf.append(String.valueOf(latest_rec.rx-previous_rec.rx));
      buf.append(")");
    }

    buf.append(", ");
    buf.append(String.valueOf(latest_rec.tx));
    buf.append(" sent");

    if (previous_rec!=null) {
      buf.append(" (delta=");
      buf.append(String.valueOf(latest_rec.tx-previous_rec.tx));
      buf.append(")");
    }

    rows.add(buf.toString());
  }
}
```

Since the lines created by `emitLog()` start with the package name, and since we are sorting those before dumping them to LogCat, they appear in LogCat in sorted order by package name.

Using TrafficMonitor

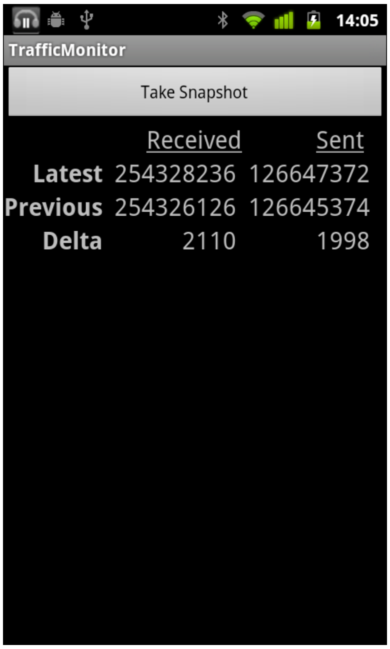
Running the activity gives you the initial received and sent counts (in bytes):



	Received	Sent
Latest	254326126	126645374
Previous		
Delta		

Figure 11. The TrafficMonitor sample application, as initially launched

Tapping Take Snapshot grabs a second snapshot and compares the two:



	Received	Sent
Latest	254328236	126647372
Previous	254326126	126645374
Delta	2110	1998

Figure 12. The TrafficMonitor sample application, after Take Snapshot was clicked

Also, LogCat will show how much was used by various apps:

```
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283):
com.amblingbooks.bookplayerpro=880 received (delta=0), 3200 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.android.browser=19045241
received (delta=0), 2375847 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283):
com.android.providers.downloads=27884469 received (delta=0), 9126 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283):
com.android.providers.telephony=2328 received (delta=0), 4912 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.android.vending=3271839
received (delta=0), 260626 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.coair.mobile.android=887425
received (delta=0), 81366 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.commonsware.android.browser1=262553 received (delta=0), 7286 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.dropbox.android=6189833
received (delta=0), 4298 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.evernote=3471398 received
(delta=0), 742178 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
```

```
com.google.android.apps.genie.geniewidget=358816 received (delta=0), 17775 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.googlevoice=103255 received (delta=0), 35559 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.maps=28440829 received (delta=0), 1230867 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.backup=51320
received (delta=0), 49041 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.gm=10915084
received (delta=0), 14428803 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.googlequicksearchbox=37817 received (delta=0), 12554 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.syncadapters.contacts=1955990 received (delta=0), 714893 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.voicesearch=67948 received (delta=0), 121908 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.youtube=3128
received (delta=0), 2792 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.howcast.android.app=2250407
received (delta=0), 26727 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.rememberthemilk.MobileRTM=6836605 received (delta=0), 2902904 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.tripit=109499 received
(delta=0), 50060 sent (delta=0)
```

Other Ways to Employ TrafficStats

Of course, there are more ways you could use `TrafficStats` than simply having an activity to report them on a button click. `TrafficMonitor` is merely a demonstration of using the class and providing a lightweight way to get value out of that data. Depending upon your application's operations, though, you may wish to consider using `TrafficStats` in other ways, in your production code or in your test suites.

In Production

If your app is a bandwidth monitor, the need to use `TrafficStats` is obvious. However, even if your app does something else, you may wish to use `TrafficStats` to understand what is going on in terms of Internet access within your app or on the device as a whole.

For example, you might want to consider bandwidth consumption to be a metric worthy of including in the rest of the "analytics" you generate from your app. If you are using services like Flurry to monitor which activities get used and so on, you might consider also logging the amount of bandwidth your application consumes. This not only gives you much more "real world" data than you will be able to collect on your own, but it may give you ideas of how users are using your application beyond what the rest of your metrics are reporting.

Another possibility would be to include your app's bandwidth consumption in error logs reported via libraries like ACRA. Just as device particulars can help identify certain bug report patterns, perhaps certain crashes of your app only occur when users are using a lot of bandwidth in your app, or using a lot of bandwidth elsewhere and perhaps choking your own app's Internet access.

The chapter on bandwidth mitigation strategies will also cover a number of uses of `TrafficStats` for real-time adjustment of your application logic.

During Testing

You might consider adding `TrafficStats`-based bandwidth logging for your application in your test suites. While individual tests may or may not give you useful data, you may be able to draw trendlines over time to see if you are consuming more or less bandwidth than you used to. Take care to factor in that you may have changed the tests, in addition to changing the code that is being tested.

From a JUnit-based unit test suite, measuring bandwidth consumption is not especially hard. You can bake it into the `setUp()` and `tearDown()` methods of your test cases, either via inheritance or composition, and log the output to a file or LogCat.

From an external test engine, like monkeyrunner or NativeDriver, recording bandwidth usage is more tricky, because your test code is not running on the device or emulator. You may have to include a

`BroadcastReceiver` in your production code that will log bandwidth usage and trigger that code via the `am broadcast` shell command.

CHAPTER 7

Measuring Bandwidth Consumption

The first step towards addressing bandwidth concerns is to get a better picture of how much bandwidth you are actually consuming, when, and under what conditions. Only then will you be able to determine where your efforts need to be applied and whether those efforts are actually giving you positive results. This chapter will examine a handful of ways you can determine how much bandwidth you are really using in your application.

On-Device Measurement

Many times, you are best served by measuring your bandwidth consumption right on the device itself:

- This is your only option for gathering bandwidth metrics from copies of your app in end users' hands, unless they invite you to their home or office and have you sniff on their personal network, which seems unlikely
- This is your only option for gathering bandwidth metrics when you are using mobile data plans (e.g., 3G) instead of WiFi, since you probably do not control the wireless telecommunications infrastructure in your area
- This is your simplest option for tying bandwidth metrics to events within your app or occurring on the device

- This is your only option for using bandwidth metrics to adjust your application behavior in real time, in addition to using the metrics to learn how best to adjust your code in future updates to the app

Hence, in addition to perhaps other off-device techniques, you really should consider one of the on-device approaches outlined in the following sections.

Yourself, via TrafficStats

The preceding chapter outlined how to use the `TrafficStats` class to collect metrics on the bandwidth consumed by applications (including yours) and for the device as a whole. This gives you the most flexibility, because you can write your own code to collect whatever portion of this data you need. It can address all of the bullets shown above, for example.

It is not perfect, though:

- It requires you to write your own code, adding yet more work to your plate
- Per-UID traffic data may or may not be available, depending upon the device

Existing Android Applications

If you do not want to write code to use `TrafficStats`, there are various applications on the Android Market that can report that data to you, much along the lines of how `TrafficMonitor` does. Here are some notes about a few free ones tested by the author:

- Network Traffic Detail (v. 1.3) works, but does not consider that bandwidth is only reported per UID, not per application. As a result, it reports the same traffic multiple times, one for each application sharing a UID.
- Traffic Monitor (v. 2.4.2) advertises itself as an application, but does not put an icon in the launcher for it, forcing you to install an app

widget instead in order to get to the actual application. While it reports device-level bandwidth, and it has a task manager, the task list does not report bandwidth for those tasks.

- Bandwidth Monitor (v. 1.0.6) works and is perhaps incrementally easier to use than the other alternatives, though its touted bar chart of bandwidth consumption lacks any indicator of the value of the Y axis.

There are certainly others on the Market today and more will show up over time. For your own use, these sorts of apps may be very helpful. However, since you control nothing over what is collected and how (and, in the case of some, even when it is collected), it may be difficult for you to get a solid grasp on where your code is consuming bandwidth this way.

There are also various apps that provide more in the way of packet-sniffing capability. However, these require you to root your phone and run the app with root privileges.

Off-Device Measurement

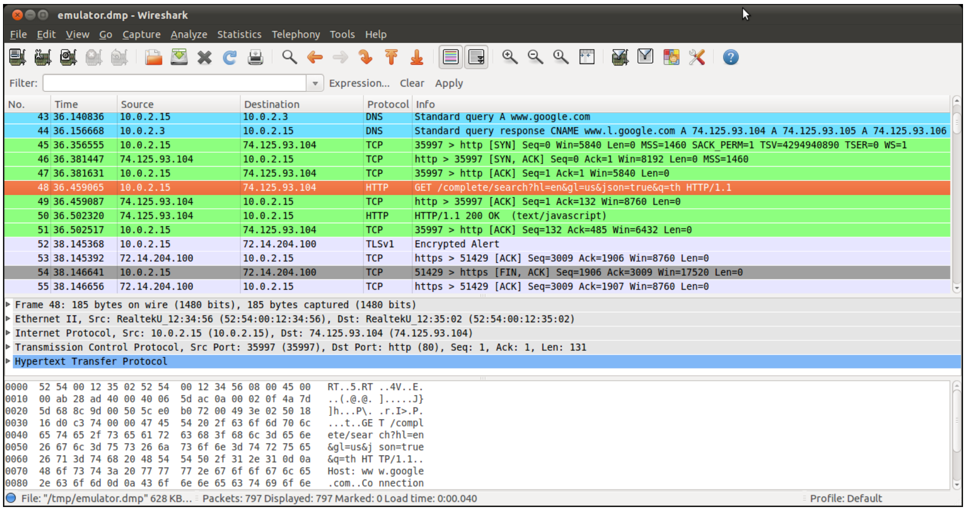
The biggest limitation of `TrafficStats` is that it only gives you gross metrics: numbers of bytes, packets, and so on. Sometimes, that is not enough to help you understand why those bytes, packets, and so on are actually being sent or received. Sometimes, it would be nice to understand the traffic in more detail, from the ports and IP addresses to perhaps the actual data being transmitted. For obvious security reasons, this is not something an ordinary Android SDK application can do. However, there are techniques for accomplishing this, mostly for use over WiFi in your own home or office network. Some of these are outlined in the following sections.

Wireshark

Wireshark, formerly known as Ethereal, is perhaps the world's leading open source network traffic analyzer and packet inspector. Using it, you can learn in great detail what is going on with your local network. And, Android provides additional options for you to leverage Wireshark to make sense of application behavior. Wireshark is available for Linux, OS X, and Windows.

There is a lightly-documented `-tcpdump` switch available on the Android emulator. If you launch the emulator from the command line with that switch (plus `-avd` to identify the AVD file you want to use), all network access is dumped to your specified log file. You can then load that data into Wireshark for analysis, via File|Open from the main menu.

For example, here is a screenshot of Wireshark examining data from such an emulator dump file, in which the emulator was used to conduct a Google search:


emulator.dmp - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression... Clear Apply

```
No.  Time       Source          Destination     Protocol Info
  43 36.140836  10.0.2.15       10.0.2.3        DNS      Standard query A www.google.com
  44 36.156668  10.0.2.3        10.0.2.15       DNS      Standard query response CNAME www.l.google.com A 74.125.93.104 A 74.125.93.105 A 74.125.93.106
  45 36.356555  10.0.2.15       74.125.93.104   TCP      35997 > http [SYN] Seq=0 Win=5840 Len=0 MSS=1460 SACK_PERM=1 TSV=4294940890 TSER=0 WS=1
  46 36.381447  74.125.93.104   10.0.2.15       TCP      http > 35997 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460
  47 36.381631  10.0.2.15       74.125.93.104   TCP      35997 > http [ACK] Seq=1 Ack=1 Win=5840 Len=0
  48 36.459065  10.0.2.15       74.125.93.104   HTTP     GET /complete/search?hl=en&gl=us&json=true&q=th HTTP/1.1
  49 36.459087  74.125.93.104   10.0.2.15       TCP      http > 35997 [ACK] Seq=1 Ack=132 Win=8760 Len=0
  50 36.502320  74.125.93.104   10.0.2.15       HTTP     HTTP/1.1 200 OK  (text/javascript)
  51 36.502517  10.0.2.15       74.125.93.104   TCP      35997 > http [ACK] Seq=132 Ack=485 Win=6432 Len=0
  52 38.145368  10.0.2.15       72.14.204.100   TLSv1    Encrypted Alert
  53 38.145392  72.14.204.100   10.0.2.15       TCP      https > 51429 [ACK] Seq=3009 Ack=1906 Win=8760 Len=0
  54 38.146641  10.0.2.15       72.14.204.100   TCP      51429 > https [FIN, ACK] Seq=1906 Ack=3009 Win=17520 Len=0
  55 38.146656  72.14.204.100   10.0.2.15       TCP      https > 51429 [ACK] Seq=3009 Ack=1907 Win=8760 Len=0

▸ Frame 48: 185 bytes on wire (1480 bits), 185 bytes captured (1480 bits)
▸ Ethernet II, Src: RealtekU_12:34:56 (52:54:00:12:34:56), Dst: RealtekU_12:35:02 (52:54:00:12:35:02)
▸ Internet Protocol, Src: 10.0.2.15 (10.0.2.15), Dst: 74.125.93.104 (74.125.93.104)
▸ Transmission Control Protocol, Src Port: 35997 (35997), Dst Port: http (80), Seq: 1, Ack: 1, Len: 131
▸ Hypertext Transfer Protocol

0000  52 54 00 12 35 02 52 54  00 12 34 56 08 00 45 00   RT..5.RT ..4V..E.
0010  00 ab 28 ad 40 00 40 06  5d ac 0a 00 02 0f 4a 7d   ..(.@.@. ].....J}
0020  5d 68 8c 9d 00 50 5c e0  b0 72 00 49 3e 02 50 18   ]h...P\. .r.I>.P.
0030  16 d0 c3 74 00 00 47 45  54 20 2f 63 6f 6d 70 6c   ...t..GE T /compl
0040  65 74 65 2f 73 65 61 72  63 68 3f 68 6c 3d 65 6e   ete/sear ch?hl=en
0050  26 67 6c 3d 75 73 26 6a  73 6f 6e 3d 74 72 75 65   &gl=us&j son=true
0060  26 71 3d 74 68 20 48 54  54 50 2f 31 2e 31 0d 0a   &q=th HT TP/1.1..
0070  48 6f 73 74 3a 20 77 77  77 2e 67 6f 6f 67 6c 65   Host: ww w.google
0080  2e 63 6f 6d 0d 0a 43 6f  6e 6e 65 63 74 69 6f 6e   .com..Co nnection

File: "/tmp/emulator.dmp" 628 KB...   Packets: 797 Displayed: 797 Marked: 0 Load time: 0:00.040   Profile: Default
```

Figure 13. Wireshark examining captured emulator packets

This screenshot shows an HTTP request in the highlighted line in the list, with the hex and ASCII contents of the request shown in the bottom pane.

In terms of using Wireshark to monitor traffic from actual hardware, that is indubitably possible. However, WiFi packet collection is a tricky process with Wireshark, being very dependent upon operating system and possibly even the WiFi adapter chipset. You also get much lower-level information, making it a bit more challenging to figure out what is going on. Attempting to cover all of this is well beyond the scope of this book and the author's Wireshark expertise.

Networking Hardware

Sophisticated firewalls sometimes have packet tracing/sniffing capability. In this case, "sophisticated" does not necessarily mean "expensive", as open source router/firewall distributions, like OpenWrt, can be used for this sort of work. In this case, the router captures the packets and, in many cases, routes them to Wireshark for analysis. Some might offer on-board analysis (e.g., Web interface to packet capture logs).

This is particularly useful on a Windows wireless network. Wireshark has limits, imposed by Windows, that cause some problems when trying to capture WiFi packets. By offloading the packet capture to networking hardware, those limits can be bypassed.

CHAPTER 8

Being Smarter About Bandwidth

Given that you are collecting metrics about bandwidth consumption, you can now start to determine ways to reduce that consumption. You may be able to permanently reduce that consumption (at least on a per-operation basis). You may be able to shunt that consumption to times or networks that the user prefers. This chapter reviews a variety of means of accomplishing these ends.

Bandwidth Savings

The best way to reduce bandwidth consumption is to consume less bandwidth.

(in other breaking news, water is wet)

In recent years, developers have been able to be relatively profligate in their use of bandwidth, pretty much assuming everyone has an unlimited high-speed Internet connection to their desktop or notebook and the desktop or Web apps in use on them. However, those of us who lived through the early days of the Internet remember far too well the challenges that dial-up modem accounts would present to users (and perhaps ourselves). Even today, as Web apps try to "scale to the Moon and back", bandwidth savings becomes important not so much for the end user, but for the Web app host, so its own bandwidth is not swamped as its user base grows.

Fortunately, widespread development problems tend to bring rise to a variety of solutions – a variant on the "many eyes make bugs shallow" collaborative development phenomenon. Hence, there are any number of tried-and-true techniques for reducing bandwidth consumption that have had use in Web apps and elsewhere. Many of these are valid for native Android apps as well, and a few of them are profiled in the following sections.

Classic HTTP Solutions

Trying to get lots of data to fit on a narrow pipe – whether that pipe is on the user's end or the provider's end – has long been a struggle in Web development. Fortunately, there are a number of ways you can leverage HTTP intelligently to reduce your bandwidth consumption.

GZip Encoding

By default, HTTP requests and response are uncompressed. However, you can enable GZip encoding and thereby request that the server compress its response, which is then decompressed on the client. This trades off CPU for bandwidth savings and therefore needs to be done judiciously.

Enabling GZip compression is a two-step process:

1. Adding the `Accept-Encoding: gzip` header to the HTTP request
2. Determine if the response was compressed and, if so, decompressing it

Bear in mind that the Web server may or may not honor your GZip request, for whatever reason (e.g., response is too small to make it worthwhile).

For example, using the HttpClient library in Android, you could add the header on the request:

```
HttpGet get=new HttpGet(url);

get.addHeader("Accept-Encoding", "gzip");

// rest of configuration here, if any

// execute the request given an HttpClient object named client
HttpResponse response=client.execute(get);
```

Then, you can check the response and get a valid `InputStream` for either the compressed or the not-compressed cases:

```
// assumes HttpResponse response as in above code snippet

InputStream stream=response.getEntity().getContent();
Header enc=response.getFirstHeader("Content-Encoding");

if (enc!=null && enc.getValue().equalsIgnoreCase("gzip")) {
 stream=new GZIPInputStream(stream);
}

// at this point, stream will work for either encoding
```

Equivalents exist for using `HttpUrlConnection`, if you prefer to use that HTTP API in Android.

If-Modified-Since / If-None-Match

Of course, avoiding a download offers near-100% compression. If you are caching data, you can take advantage of HTTP headers to try to skip downloads that are the same content as what you already have, specifically `If-Modified-Since` and `If-None-Match`.

An HTTP response can contain either a `Last-Modified` header or an `ETag` header. The former will contain a timestamp and the latter will contain some opaque value. You can store this information with the cached copy of the data (e.g., in a database table). Later on, when you want to ensure you have the latest version of that file, your HTTP GET request can include an `If-Modified-Since` header (with the cached Last-Modified value) or an `If-None-Match` header (with the cached `ETag` value). In either case, the server should return either a `304` response, indicating that your cached copy is up to date, or a `200` response with the updated data. As a result, you avoid the download entirely (other than HTTP headers) when you do not need the updated data.

For example, using HttpClient, you can check for the existence of an `ETag` header in an HTTP response:

```
HttpGet get=new HttpGet(url);

// execute the request given an HttpClient object named client
HttpResponse response=client.execute(get);
Header etag=response.getFirstHeader("ETag");

if (etag!=null) {
 // cache this
}

// process the download
```

On subsequent requests, you can add the `If-None-Match` header and handle both cases:

```
HttpGet get=new HttpGet(url);

get.addHeader("If-None-Match", etag);

// execute the request given an HttpClient object named client
HttpResponse response=client.execute(get);
int sc=response.getStatusLine().getStatusCode();

if (sc!=HttpStatus.SC_NOT_MODIFIED) {
 // cache invalid, so process the download and, perhaps, grab fresh ETag
}
```

Using `Last-Modified` and `If-Modified-Since` is mostly a matter of switching headers. And, once again, there are equivalent ways to use these headers with `HttpUrlConnection`.

Binary Payloads

While XML and JSON are relatively easy for humans to read, that very characteristic means they tend to be bloated in terms of bandwidth consumption. There are a variety of tools, such as Google's Protocol Buffers and Apache's Thrift, that allow you to create and parse binary data structures in a cross-platform fashion. These might allow you to transfer the same data that you would in XML or JSON in less space. As a side benefit, parsing the binary responses is likely to be faster than parsing XML or JSON. Both of these tools involve the creation of an IDL-type file to describe the data structure, then offer code generators to create Java classes (or equivalents for other languages) that can read and write such structures, converting them into platform-neutral on-the-wire byte arrays as needed.

Minification

If you are loading JavaScript or CSS into a `WebView`, you should consider standard tricks for compressing those scripts, collectively referred to as "minification". These techniques eliminate all unnecessary whitespace and such from the files, rename variables to be short, and otherwise create a syntactically-identical script that takes up a fraction of the space. There are services like box.js that can even aggregate several scripts into one file and minify them, to further reduce HTTP overhead.

Push versus Poll

Another way to consume less bandwidth is to only make the requests when it is needed. For example, if you are writing an email client, the way to use the least bandwidth is to download new messages only when they exist, rather than frequently polling for messages.

Off the cuff, this may seem counter-intuitive. After all, how can we know whether or not there are any messages if we are not polling for them?

The answer is to use a low-bandwidth push mechanism. The quintessential example of this is C2DM, the Cloud-to-Device Messaging system, available for Android 2.2 and newer. This service from Google allows your application to subscribe to push notifications sent out by your server. Those notifications are delivered asynchronously to the device by way of Google's own servers, using a long-lived socket connection. All you do is register a `BroadcastReceiver` to receive the notifications and do something with them.

For example, Remember the Milk – a task management Web site and set of mobile apps – uses C2DM to alert the device of task changes you make through the Web site. Rather than the Remember the Milk app having to constantly poll to see if tasks were added, changed, or deleted, the app simply waits for C2DM events.

You could create your own push mechanism, perhaps using a WebSocket or Comet-style long-poll technique. The downside is that you will need a service in memory all of the time to manage the socket and thread that monitors it. If you only need this while your service is in memory for other reasons, that is fine. However, keeping a service in memory 24x7 has its own set of issues, not the least of which is that users will tend to smack it down using a "task killer" or the Manage Services screen in the Settings app.

C2DM itself is covered in greater detail in The Busy Coder's Guide to Advanced Android Development.

Thumbnails and Tiles

A general rule of thumb is: don't download it until you really need it.

Sometimes, you do not know if you really need a particular item until something happens in the UI. Take a `ListView` displaying thumbnails of album covers for a music app. Assuming the album covers are not stored locally, you will need to download them for display. However, which covers you need varies based upon scrolling. Downloading a high-resolution album cover that might get tossed in a matter of milliseconds (after an expensive rescale to fit a thumbnail-sized space) is a waste of bandwidth.

In this case, either the album covers are something you control on the server side, or they are not. If they are, you can have the server prepare thumbnails of the covers, stored at a spot that the app can know about (e.g., `.../cover.jpg` it is `.../thumbnail.jpg`). The app can then download thumbnails on the fly and only grab the full-resolution cover if needed (e.g., user clicks on the album to bring up a detail screen). If you do not control the album covers, this option might still be available to you if you can run your own server for the purposes of generating such thumbnails.

You can see a similar effect with the map tiles in Google Maps. When zooming out, the existing map tiles are scaled down, with placeholders (the gridlines) for the remaining spots, until the tiles for those spots are downloaded. When zooming in, the existing map tiles are scaled up with a slight blurring effect, to give the user some immediate feedback while the full set of more-detailed tiles is downloaded. And, if the user pans, you once again get placeholders while the tiles for the newly uncovered areas are downloaded. In this fashion, Google Maps is able to minimize bandwidth consumption by giving users partial results immediately and back-filling in the final results only when needed. This same sort of approach may be useful with your own imagery.

Collaborative Bandwidth

For some common services, perhaps sharing is the best option to reduce bandwidth usage.

For example, consider Twitter. It is entirely possible that a user might have multiple applications all polling and downloading the user's timeline:

- A built-in Twitter app that the user does not like, but cannot uninstall
- A regular Twitter app that the user employs for normal stuff
- A separate Twitter app widget, because the other Twitter apps on the device either lack an app widget or the user does not like it
- Yet another application that uses Twitter as one of several data sources (e.g., monitoring for references to certain keywords, such as a company name, across multiple social networks)

In an ideal world, all of these apps would use one common engine that handles collecting the tweets and making them available – securely – to the other applications. This would dramatically cut bandwidth by eliminating redundant polling.

If your data source is used by other applications, consider reaching out to those developers and creating a common engine, perhaps using a `ContentProvider` for data sharing, an `IntentService` or sync provider for collecting the data, plus common activities for preferences. Distribute the code to all of the development teams as an Android library project. Ship these components disabled in your manifest, enabling them if you cannot find another implementation on the device, indicating that you are the only one of this "application family" installed. If you do find another implementation, use that one instead of your own. There are certainly issues to be dealt with here (e.g., what if the user uninstalls the app that the others are depending upon), but it is worth considering for shared development costs as well as shared bandwidth.

Bandwidth Shaping

Sometimes, you have no ability to reduce the bandwidth itself. Perhaps you do not control both ends of the communications pipeline. Perhaps the data you are trying to exchange is already compressed (e.g., downloading an MP4 video). Perhaps some of the techniques in the preceding section were unavailable to you (e.g., cannot route data through third-party servers like Google's for C2DM).

There still may be ways for you to help your users, by shaping your bandwidth use. Rather than just blindly doing whatever you want whenever you want, you learn what the *user* wants and what *other applications* want and tailor your bandwidth use on the fly to match those needs. The following sections outline some ways of achieving this.

Driven by Preferences

If you are consuming enough bandwidth that this chapter is relevant to you, you probably are consuming enough bandwidth that you should be asking the user how best to consume that bandwidth. After all, they are the one paying the price – in time as well as money – for that consumption.

The following sections present some possible strategies for preference-based bandwidth shaping.

Budgets

One strategy is for the user to give you a budget (e.g., 20MB/day) and for you to stick within that budget.

Collecting the budget is fairly easy – just use `SharedPreferences`. Either use a `ListPreference` with likely budget value or an `EditTextPreference` and a bit of validation for a free-form budget amount.

Next, you will need to have some idea how much bandwidth any given network operation will consume. For some things, this might be an estimate based on your experiments as a developer, or perhaps it is based on historical averages for this user and type of operation. For example, a "podcatcher" (feed reader designed to download podcast episodes) should have some idea how big a given RSS or Atom feed download should be. In some cases, it might be worthwhile to get a better estimate – for example, the podcatcher might use an HTTP `HEAD` request to determine the size of the MP3 or OGG file before deciding whether to download it.

Then, you need to be keeping track of your budget. This could be a simple flat file with the initial `TrafficStats` bandwidth values for your process. Re-initialize that file on the first network operation of the day (or whatever period you chose for your budget). Before doing another network operation, compare the current `TrafficStats` values with the initial ones and see how close you are to the budget. If the new network operation will exceed the budget, skip the operation, perhaps putting it in a work queue to perform in the next budget. You might even hold a reserve for certain types of operations. For example, the podcatcher might ensure there is at least 10% of the budget available for downloading the feeds, even if it means putting a podcast on the queue for download tomorrow. That way, you can present to the user the latest podcast information, with icons indicating which are downloaded and which are queued for download – the user might be able to then request to override the budget and download something on demand.

For devices that lack per-UID `TrafficStats` support, you will have to "fake it" a bit. Use your own calculations of how much bandwidth each operation consumes and track that information, even if you wind up missing out on some bytes here or there.

Connectivity

If the user might not care how much bandwidth you consume, so long as it is un-metered bandwidth, you might include a `CheckBoxPreference` to indicate if large network operations should be limited to WiFi and avoid mobile data.

You could then use `ConnectivityManager` and `getActiveNetworkInfo()` to see what connection you have before performing a network operation. If it is a background operation (e.g., the podcatcher checking for new podcasts every hour), if the network is not the desired one, you can skip the operation or put it on a work queue for re-trying later. If it is a foreground operation (e.g., the user clicked a "refresh" menu choice), you could pop up a confirmation `AlertDialog` to warn the user that they are on mobile data – perhaps this time they are interested in doing the operation anyway.

Another approach for handling the background operations is to register a `BroadcastReceiver` for the `CONNECTIVITY_ACTION` broadcast (defined on `ConnectivityManager`). If the connectivity switches to mobile data, cancel your outstanding `AlarmManager` alarms; if connectivity switches to WiFi, re-enable those alarms.

Of course, you should also consider monitoring the background data setting – the global Settings checkbox indicating whether background network operations are allowed. On `ConnectivityManager`, `getBackgroundDataSetting()` tells you the state of this checkbox, and `ACTION_BACKGROUND_DATA_SETTING_CHANGED` allows you to set up a `BroadcastReceiver` to watch for changes in its state.

Windows

If your user is less concerned about the bandwidth or the network, but does care about the time of day (e.g., does not want your application consuming significant bandwidth when they might be getting a VOIP call), you could offer preferences for that as well. Cook up a `TimePreference` (you can find an example of one in the source for *Android Programming Tutorials*) and use that to collect start and stop times for the high-bandwidth window. Then, set up alarms with `AlarmManager` for those points in time. The alarm for the start time of the window sets up a third alarm with your regular polling interval. The alarm for the stop time of the window cancels the polling interval alarm.

Driven by Other Usage

If your network I/O is part of a foreground application, one presumes that you are the most important thing in the user's life right now. Or, at least, the most important thing on the user's phone right now. Hence, what other applications might want to do with the Internet connection is not a major concern.

If, however, your network I/O is part of a background operation, it might be nice to try to avoid doing things that might upset the user. If the user is watching streaming video or is on a VOIP call or otherwise is aware of bandwidth changes, the bandwidth you use might impact the user in ways that the user will not appreciate very much. This is unlikely to be a big problem for small operations (e.g., downloading a 1KB JSON file), but larger operations (e.g., downloading a 5MB podcast) might be more noticeable.

You can use `TrafficStats` to help here. Before doing the actual network I/O, grab the current traffic data, wait a couple of seconds, and compare the latest to the previous values. If little to no bandwidth was consumed during that period, assume it is safe and go ahead and do your work. If, however, a bunch of bandwidth was consumed, you might want to consider:

- Skipping this polling cycle and trying again later, or

- Adding a one-off alarm using set() on `AlarmManager` to give you control again in a minute, with the current traffic data packaged as an extra on the Intent, so you can make a decision after a bigger sample size of bandwidth consumption, or
- Adding an entry in a persistent work queue, so you know later on to try again if bandwidth contention has improved

You could try to get more sophisticated, by using `ActivityManager` and the per-UID values from `TrafficStats` to see if it is a foreground application that is the one consuming the bandwidth. It is unclear how reliable this will be, both in determining who is consuming the bandwidth (again, per-UID traffic is not available on many devices) and in avoid user angst. It may be simpler just to assume the worst and side-step your I/O until the other apps have quieted down.

PART III – Memory Performance

CHAPTER 9

Issues with Memory

RAM. Developers nowadays are used to having lots of it, and a virtual machine capable of using as much of it as exists (and more, given swap files and page files).

"Graybeards" – like the author of this book – distinctly remember a time when we had 16KB of RAM and were happy for it. Such graybeards would also appreciate it if you would get off their respective lawns.

Android comes somewhere in the middle. We have orders of magnitude more RAM than, say, the TRS-80 Model III. We do not have nearly as much RAM as does the modern notebook, let alone a Web server. As such, it is easy to run out of RAM if you do not take sufficient care.

This part of the book examines memory-related issues. These are not to be confused with any memory-related issues inherent to graybeards.

You Are in a Heap of Trouble

When we think of "memory" and Java-style programming, the primary form of memory is the heap. The heap holds all of our Java objects – from an `Activity` to a widget to a `String`.

Traditional Java applications have an initial heap size determined by the virtual machine, possibly configured via command-line options when the program was run. Traditional Java applications can also request additional memory from the OS, up to some maximum, also configurable.

Android applications have the same basic structure, with very limited configurability and much lower maximums than you might expect.

Older Android devices, particularly those with HVGA screens like the T-Mobile G1, tend to have a maximum of 16MB of heap space. Newer Android phones with higher-resolution screens might have 24MB (Motorola DROID) or 32MB (Nexus One) of heap space. Tablets might have 48MB of heap space.

This heap limit can be problematic. For example, each widget or layout manager instance takes around 1KB of heap space. This is why `AdapterView` provides the hooks for view recycling – we cannot have a `ListView` with literally thousands of row views without potentially running out of heap.

API Level 11 now supports applications requesting a "large heap". This is for applications that specifically need tons of RAM, such as an image editor to be used on a tablet. This is not for applications that run out of heap due to leaks or sloppy programming. Bear in mind that users will feel effects from large-heap applications, in that their other applications will be kicked out of memory more quickly, possibly irritating them. Also, garbage collection on large-heap applications runs more slowly, consuming more CPU time. To enable the large heap, add `android:largeHeap="true"` to the `<application>` element of your manifest. You can call `getLargeMemoryClass()` on `ActivityManager` to learn how large your "large heap" actually is.

Warning: Contains Graphic Images

However, the most likely culprit for `OutOfMemoryError` messages are bitmaps. Bitmaps take up a remarkable amount of heap space. Developers often look at the size of a JPEG file and think that "oh, well, that's only a handful of KB", without taking into account:

- the fact that most image formats, like JPEG and PNG, are compressed, and Android needs the uncompressed image to know what to draw
- the fact that each pixel may take up several bytes (2 bytes per pixel for `RGB_565`, 3 bytes per pixel for `RGB_888`)
- what matters is the resolution of the bitmap in its original form, as much (if not more) than the size in which it will be rendered – an 800x480 image displayed in an 80x48 `ImageView` still consumes 800x480 worth of pixel data
- there are an awful lot of pixels in an image – 800 times 480 is 384,000

Android can make some optimizations, such as only loading in one copy of a Drawable resource no matter how many times you render it. However, in general, each bitmap you load takes a decent sized chunk of your heap, and too many bitmaps means not enough heap. It is not unheard of for an application to have more than half of its heap space tied up in various bitmap images.

Compounding this problem is that bitmap memory, before Honeycomb, was difficult to measure. In the actual Dalvik heap, a Bitmap would need ~80 bytes or so, regardless of image size. The actual pixel data was held in "native heap", the space that a C/C++ program would obtain via calls to `malloc()`. While this space was still subtracted from the available heap space, many diagnostic programs – such as MAT, to be examined in the next chapter – will not know about it. Android 3.0 (code-named "Honeycomb") moved the pixel data into the Dalvik heap, which will improve our ability to find and deal with memory leaks or overuse of bitmaps.

This part of the book will cover techniques to identify where you might be leaking memory and what is consuming all of your heap space if you are running out of it. We will also examine ways to avoid such leaks and be more efficient in your memory consumption, particularly with bitmaps.

In Too Deep (on the Stack)

Heap, however, is not the only possible source of memory errors. It is also possible to get an `StackOverflowError`, indicating that you have run out of stack space (or possibly that the leading Android developer support resource is down for maintenance).

In stack-based programming languages like Java, each time you call a method, some stack space is consumed. While method parameters are objects that live on the heap, the parameter references are stored on the stack, as is information about the method being invoked. References to local data members to the method or blocks inside of it are also stored on the stack.

Since these references only take up ~4 bytes each, you would think it might take a minor eternity to run out of stack space. However, the main application thread in your Android application has an 8KB stack, which means you can run out of stack space with only a couple of thousand objects on it.

Even still, it would take hundreds and hundreds of nested method invocations to put a couple of thousand objects onto the stack. In normal programming, you might only encounter this with a runaway bit of recursion, in which case no amount of stack would save you.

However, Android GUIs are fairly stack-driven. You can run out of stack space if your UI becomes too complex. More specifically, you might run out of stack space if your view hierarchy – from the root container of the Android window to the widgets inside of the containers inside of your rows inside of your `ListView` inside of your `TabHost` – gets too deep. A depth of 15 or so makes you very likely to run out of stack space somewhere along the line. So if you get the stack-space exception and the stack trace seems to be all in Andoid UI rendering code, your view hierarchy is probably too complex. In this part of the book, we will examine how to measure your view hierarchy depth and ways of trying to simplify it.

CHAPTER 10

Focus On: MAT

The Eclipse Memory Analyzer (MAT) is your #1 tool for identifying memory leaks and the culprits behind running out of heap space. Particularly when used with Honeycomb or newer versions of Android, MAT can identify:

- Who are the major sources of memory consumption, both directly (e.g., bitmaps) or indirectly (e.g., leaked activities holding onto lots of widgets)
- What is keeping objects in memory unexpectedly, defying standard garbage collection – the way that you leak memory in a managed runtime environment like Dalvik

This chapter will identify how to collect heap data for use with MAT and how to use MAT to make sense of what the heap is trying to tell us about what is going on inside of your app.

Setting Up MAT

MAT is an official Eclipse project, hosted on the Eclipse Web site. It comes in two flavors:

- A plug-in for Eclipse itself, providing a new "Memory Analysis" perspective and related tools
- A standalone version, running in the Eclipse RCP framework

Some developers may prefer the standalone version, because they run into problems when their Eclipse workspaces have too many plugins. Some developers may prefer the integrated version, because two Eclipse-based apps would consume too much RAM. With MAT, you have your choice.

There is a traditional download link to get the standalone edition. As with other Eclipse plug-ins, you will need to add the MAT update site to Eclipse – for example, in Eclipse Galileo:

- Choose Help|Install New Software... from the main menu
- Click the Add... button in the upper-right corner of the dialog, fill in `http://download.eclipse.org/mat/1.1/update-site/` as the Location and whatever name you want, then click OK
- Choose Memory Analyzer for Eclipse IDE and complete the rest of the new-software wizard

Getting Heap Dumps

The first step to analyzing what is in your heap is to actually get your hands on what is in your heap. This is referred to as creating a "heap dump" – what amounts to a log file containing all your objects and who points to what.

There are multiple ways of obtaining a heap dump, depending on your tools and use cases. Note that you will find some blog post and the like indicating you can create a heap dump via the `adb shell kill` command, but this has been disabled in newer versions of Android.

From DDMS

You can get a heap dump any time you want from DDMS, using either the DDMS perspective or the standalone DDMS utility.

In the device-and-process tree (the Devices tool in Eclipse), you will find a toolbar button that looks like a half-empty can with a downward-pointing arrow:

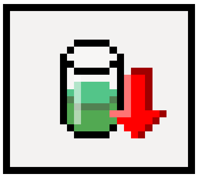


Figure 14. The icon used for the "Dump HPROF File" toolbar button

Clicking this – after choosing your desired processs – DDMS will create a heap dump for you. However, the process varies at this point, depending on whether you are using the DDMS perspective in Eclipse or standalone DDMS.

DDMS Perspective

Once you click the toolbar button for the heap dump, DDMS will create the dump for you, in a file generated in your development machine's temporary-files directory (e.g., `/tmp`). If you wish to save this dump for some reason, you will want to rename it and move it to some other location.

Standalone DDMS

Once you click the toolbar button for the heap dump, DDMS will create the dump for you, in a file chosen by you via your platform's standard file-save dialog.

Then, however, you will need to run the `hprof-conv` utility, from the `tools/` directory of your SDK, to convert the heap dump into the format that MAT will use. This is automatic if you use the DDMS perspective in Eclipse.

From Code

Another possibility is to trigger the heap dump yourself from code. The `dumpHprofData()` static method on the `Debug` class (in the `android.os` package) will write out a heap dump to the file you indicate. Since these files can be big, and since you will need to transfer them off the device or emulator, it will be best to specify a path to a file on external storage, which means that your project will need the `WRITE_EXTERNAL_STORAGE` permission.

To view the results in MAT, you will need to transfer the file to your development machine (e.g., DDMS File Manager, `adb pull`, using MTP-mounted external storage on Honeycomb).

Automating Heap Dumps in Testing

One problem with using `dumpHprofData()` is that there is no logical reason to have that code in your production app. Fortunately, you can use it from a JUnit test suite that uses the Android instrumentation framework. However, the main project, not the test project, is the one that needs `WRITE_EXTERNAL_STORAGE` – with luck, your app needs this permission anyway.

The problem then becomes a matter of figuring out where in the JUnit test suite to call `dumpHprofData()`. One strategy is simply to add it to specific test methods or test cases, if you want to have a dump at specific points. If, however, you want a dump at the end of the complete battery of tests, you will need to create your own test runner.

For example, in `MAT/Spinners` in the book's source code, you will find a near-identical clone of the same project from The Busy Coder's Guide to Advanced Android Development. It simply runs through a pathetic little test suite for an app that displays contact data in a `ListView`, driven by a `Spinner` to select what data you want to see.

The augmented version of this project adds an `HprofTestRunner` that will dump the heap at the end of the run:

```java
package com.commonsware.android.contacts.spinners;

import java.io.File;
import java.io.IOException;
import android.os.Bundle;
import android.os.Debug;
import android.os.Environment;
import android.test.InstrumentationTestRunner;

public class HprofTestRunner extends InstrumentationTestRunner {
  @Override
  public void finish(int resultCode, Bundle results) {
    try {
      Debug.dumpHprofData(new File(Environment.getExternalStorageDirectory(),
"hprof.dmp").getAbsolutePath());
    }
    catch (IOException e) {
      e.printStackTrace();
    }

    super.finish(resultCode, results);
  }
}
```

To add code at the end of a test run, simply override the `finish()` method, do your work, then chain to the superclass. Here, we create an `hprof.dmp` file out in the root of external storage. Note that the runner does not log to LogCat, which is why this code uses the classic `printStackTrace()` to dump any exceptions to the test runner's own error log.

To use the `HprofTestRunner`, you need to update the `android:name` attribute in the `<instrumentation>` element in your manifest to reference this runner class:

```xml
<?xml version="1.0" encoding="utf-8"?>
<!-- package name must be unique so suffix with "tests" so package loader
doesn't ignore us -->
<manifest android:versionCode="1"
          android:versionName="1.0"
          package="com.commonsware.android.contacts.spinners.tests"
          xmlns:android="http://schemas.android.com/apk/res/android">

  <!-- We add an application tag here just so that we can indicate that
                               this package needs to link against the
android.test library,
                               which is needed when building test cases. -->
  <application>
    <uses-library android:name="android.test.runner" />
  </application>
  <!--
                This declares that this application uses the instrumentation
test runner targeting
                the package of com.commonsware.android.database.        To run
the tests use the command:
                "adb shell am instrument -w
com.commonsware.android.database.tests/android.test.InstrumentationTestRunner"
                -->
  <instrumentation android:label="Tests for ContactSpinners"
                  android:name="com.commonsware.android.contacts.spinners.HprofT
estRunner"
                  android:targetPackage="com.commonsware.android.contacts.spinne
rs" />

</manifest>
```

Also, in your `build.xml` file for Ant, you will need to add the `test.runner` property, identifying the same class, before the `<setup/>` tag:

```
<property name="test.runner"
  value="com.commonsware.android.contacts.spinners.HprofTestRunner"/>

<setup />
```

Then, running the tests via `ant run-tests` will use your runner and will dump the HPROF file at the end of the run. You could also elect to automate retrieving the HPROF file by adding an Ant task that will use `adb pull` to retrieve the file from where it is stored.

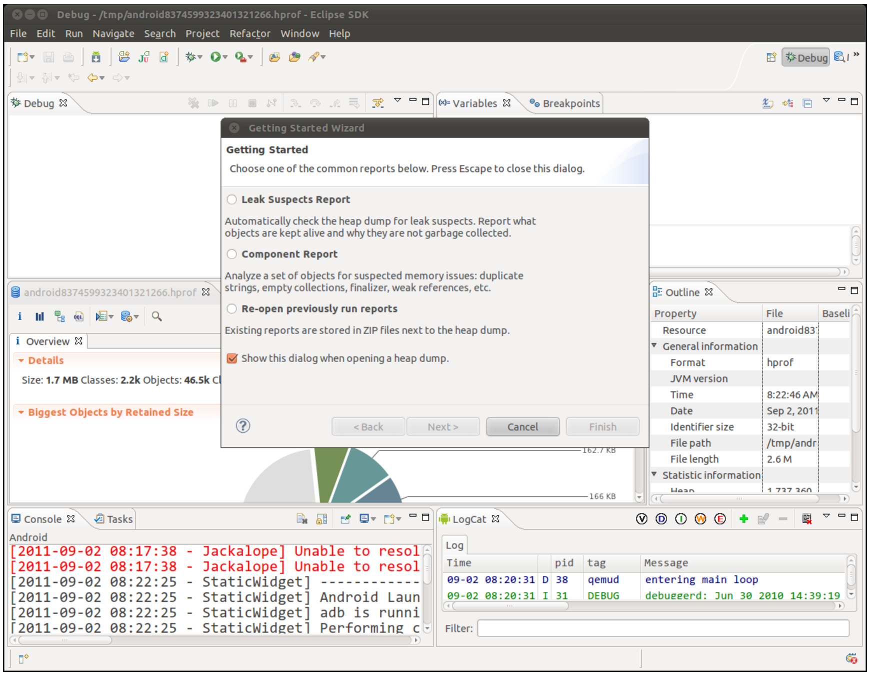
If you wish to run your tests through Eclipse, you will need to change the Instrumentation property of your test projects to point to your custom `InstrumentationTestRunner` subclass.

Basic MAT Operation

Once you have MAT installed and you have obtained a heap dump, you can start doing some analysis.

Loading Your Dump

If you used the DDMS perspective in Eclipse to create the heap dump, it should automatically pop you into MAT:


Debug - /tmp/android8374599323401321266.hprof - Eclipse SDK
File Edit Run Navigate Search Project Refactor Window Help
Debug
Variables
Breakpoints
android8374599323401321266.hprof
Overview
Details
Biggest Objects by Retained Size
162.7 KB
166 KB
Outline
Console
Tasks
LogCat

Getting Started Wizard

Getting Started

Choose one of the common reports below. Press Escape to close this dialog.

( ) **Leak Suspects Report**

Automatically check the heap dump for leak suspects. Report what objects are kept alive and why they are not garbage collected.

( ) **Component Report**

Analyze a set of objects for suspected memory issues: duplicate strings, empty collections, finalizer, weak references, etc.

( ) **Re-open previously run reports**

Existing reports are stored in ZIP files next to the heap dump.

[✓] Show this dialog when opening a heap dump.

< Back | Next > | Cancel | Finish

Size: 1.7 MB Classes: 2.2k Objects: 46.5k Cl

Property	File	Baseli
Resource	android83	
General information		
Format	hprof	
JVM version		
Time	8:22:46 AM	
Date	Sep 2, 2011	
Identifier size	32-bit	
File path	/tmp/andr	
File length	2.6 M	
Statistic information		
Heap	1 737 360	

```
Android
[2011-09-02 08:17:38 - Jackalope] Unable to resol
[2011-09-02 08:17:38 - Jackalope] Unable to resol
[2011-09-02 08:22:25 - StaticWidget] ------------
[2011-09-02 08:22:25 - StaticWidget] Android Laun
[2011-09-02 08:22:25 - StaticWidget] adb is runni
[2011-09-02 08:22:25 - StaticWidget] Performing c
```

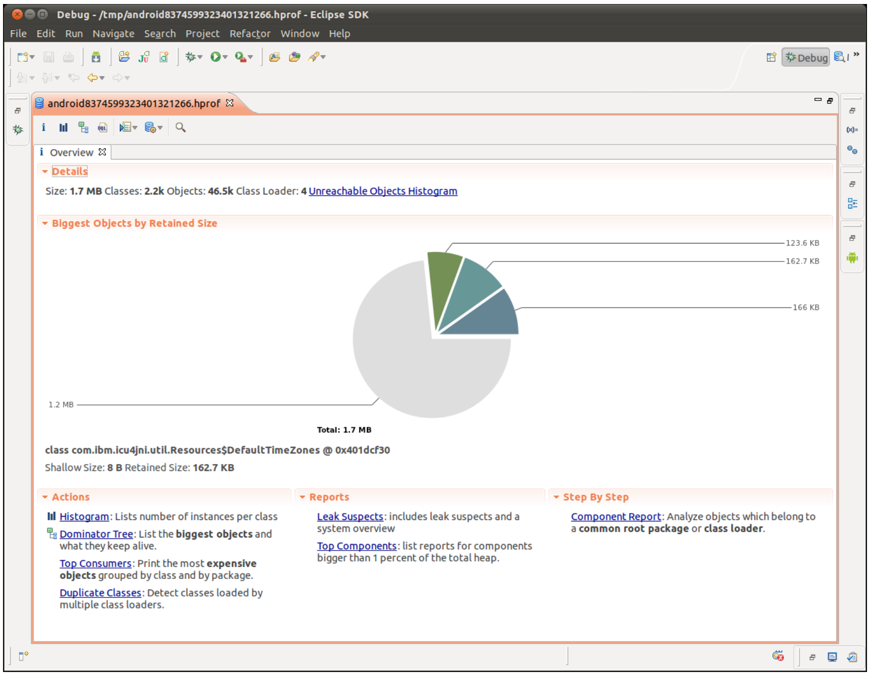
```
Log
Time                 pid  tag     Message
09-02 08:20:31  D  38   qemud   entering main loop
09-02 08:20:31  I  31   DEBUG   debuggerd: Jun 30 2010 14:39:19
Filter:
```

Figure 15. The MAT Eclipse perspective, as initially opened

If you used standalone DDMS or the code-based way of getting a heap dump, after using `hprof-conv` to create a MAT-compatible version of your dump, you can open it using the File|Open Heap Dump… menu from the Eclipse (or standalone MAT) main menu.

The first time you run MAT, you will be presented with the "Getting Started Wizard" (see above screenshot), which you can use or dismiss as you see fit.

The Overview tool gives you, well, an overview of the contents of the heap dump:


Debug - /tmp/android8374599323401321266.hprof - Eclipse SDK
File Edit Run Navigate Search Project Refactor Window Help
Debug
android8374599323401321266.hprof
Overview
123.6 KB
162.7 KB
166 KB
1.2 MB
Total: 1.7 MB

Details

Size: **1.7 MB** Classes: **2.2k** Objects: **46.5k** Class Loader: **4** Unreachable Objects Histogram

Biggest Objects by Retained Size

class com.ibm.icu4jni.util.Resources$DefaultTimeZones @ 0x401dcf30

Shallow Size: **8 B** Retained Size: **162.7 KB**

Actions

- Histogram: Lists number of instances per class
- Dominator Tree: List the **biggest objects** and what they keep alive.
- Top Consumers: Print the most **expensive objects** grouped by class and by package.
- Duplicate Classes: Detect classes loaded by multiple class loaders.

Reports

- Leak Suspects: includes leak suspects and a system overview
- Top Components: list reports for components bigger than 1 percent of the total heap.

Step By Step

- Component Report: Analyze objects which belong to a **common root package** or **class loader**.

Figure 16. The Oveview tool inside the MAT Eclipse perspective

The Overview tool also has links and toolbar buttons to get you to the other major functional areas within MAT.

Finding Your Objects

If you want to see if instances of your own classes are being kept in memory despite garbage collection, you can search for objects based upon a regular expression on the fully-qualified class name.

One way to access this is via the Histogram, reachable via a link in the Overview's Actions area or via a toolbar button:

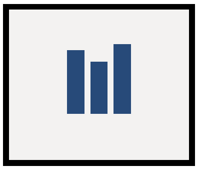
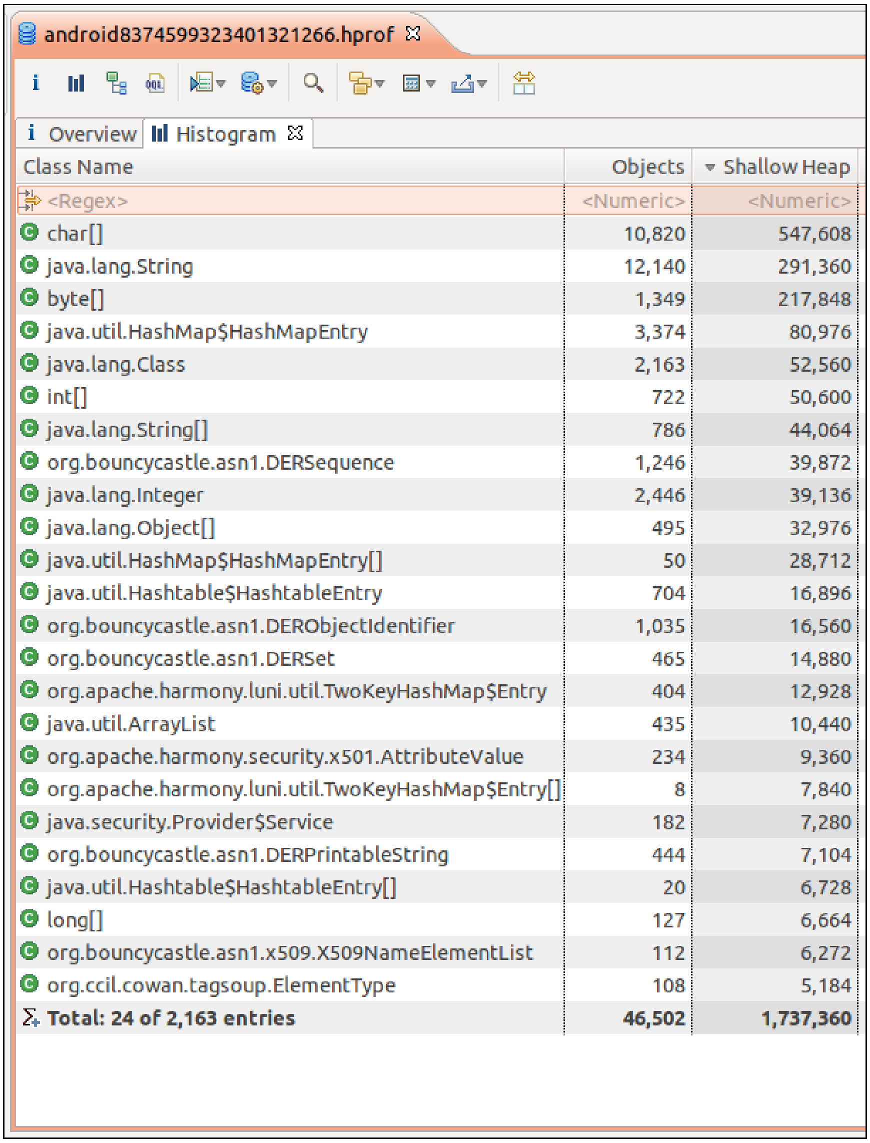


Figure 17. The icon used for the Histogram toolbar button

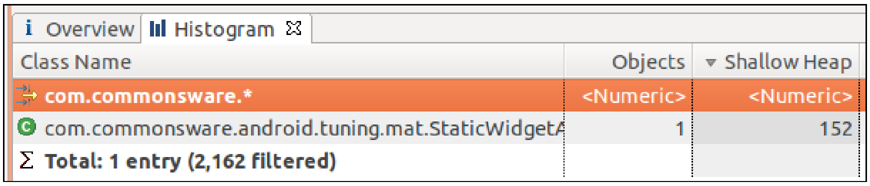
The histogram initially displays the top culprits in terms of "shallow heap" – the amount of memory those objects hold onto directly:



Class Name	Objects	Shallow Heap
<Regex>	<Numeric>	<Numeric>
char[]	10,820	547,608
java.lang.String	12,140	291,360
byte[]	1,349	217,848
java.util.HashMap$HashMapEntry	3,374	80,976
java.lang.Class	2,163	52,560
int[]	722	50,600
java.lang.String[]	786	44,064
org.bouncycastle.asn1.DERSequence	1,246	39,872
java.lang.Integer	2,446	39,136
java.lang.Object[]	495	32,976
java.util.HashMap$HashMapEntry[]	50	28,712
java.util.Hashtable$HashtableEntry	704	16,896
org.bouncycastle.asn1.DERObjectIdentifier	1,035	16,560
org.bouncycastle.asn1.DERSet	465	14,880
org.apache.harmony.luni.util.TwoKeyHashMap$Entry	404	12,928
java.util.ArrayList	435	10,440
org.apache.harmony.security.x501.AttributeValue	234	9,360
org.apache.harmony.luni.util.TwoKeyHashMap$Entry[]	8	7,840
java.security.Provider$Service	182	7,280
org.bouncycastle.asn1.DERPrintableString	444	7,104
java.util.Hashtable$HashtableEntry[]	20	6,728
long[]	127	6,664
org.bouncycastle.asn1.x509.X509NameElementList	112	6,272
org.ccil.cowan.tagsoup.ElementType	108	5,184
Total: 24 of 2,163 entries	**46,502**	**1,737,360**

Figure 18. The Histogram tab inside the MAT Eclipse perspective

To see what objects of yours might still be in the heap, you can type in a regular expression (e.g., `com.commonsware.*`) in the Regex row at the top of the table, then press <Enter> to view a filtered list of objects based upon that regular expression:



Class Name	Objects	Shallow Heap
com.commonsware.*	<Numeric>	<Numeric>
com.commonsware.android.tuning.mat.StaticWidgetA	1	152
Σ Total: 1 entry (2,162 filtered)		

Figure 19. A filtered histogram, showing com.commonsware.* objects

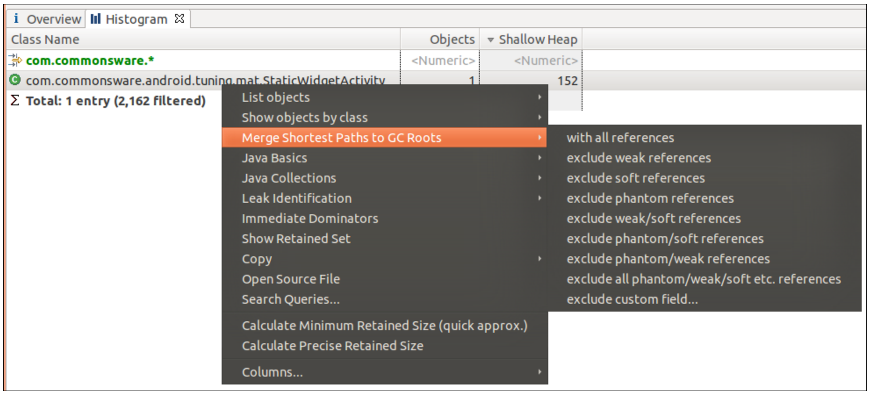
Here, we see one instance of a `com.commonsware` class is still lurking around a heap dump.

Getting Back to Your Roots

However, just because we see an object in MAT does not necessarily mean that is has been leaked. For example, this is an activity – just looking at the above screenshot does not indicate whether that activity was in the foreground, was in the background for normal reasons, or is actually leaked.

To help determine what is keeping the object in memory, you will need to trace back to the "GC roots" – the objects that are preventing our activity from being garbage collected.

To do this, you will right-click over the object in question and choose the "GC Roots" context menu choice (in the Histogram, it is "Merge Shortest Paths to GC Roots"). This will usually bring up a flyout sub-menu where you can further constrain what is reported as a root:

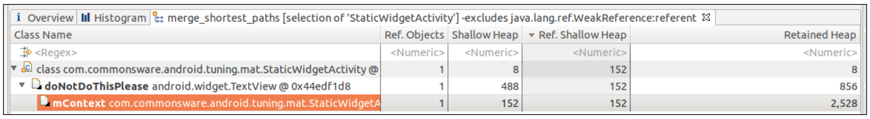

Overview
Histogram
List objects
Show objects by class
Merge Shortest Paths to GC Roots
Java Basics
Java Collections
Leak Identification
Immediate Dominators
Show Retained Set
Copy
Open Source File
Search Queries...
Calculate Minimum Retained Size (quick approx.)
Calculate Precise Retained Size
Columns...
with all references
exclude weak references
exclude soft references
exclude phantom references
exclude weak/soft references
exclude phantom/soft references
exclude phantom/weak references
exclude all phantom/weak/soft etc. references
exclude custom field...

Class Name	Objects	Shallow Heap
com.commonsware.*	<Numeric>	<Numeric>
com.commonsware.android.tuning.mat.StaticWidgetActivity	1	152
Σ Total: 1 entry (2,162 filtered)		

Figure 20. A filtered histogram, showing com.commonsware.* objects

The big filters are for "soft references" and "weak references". These refer to the `SoftReference` and `WeakReference` classes in Java, respectively. Both are ways to hold onto an object yet still allow it to be garbage collected when needed. The big difference is that an object only referenced by `WeakReference` objects can be garbage collected immediately, while an object referenced only by `SoftReference` objects (or a mix of `SoftReference` and `WeakReference` objects) should be kept around until the Dalvik VM is low on memory. Usually, you can ignore weak references, as those just indicate objects that the garbage collector has not quite detected are eligible for reclamation. Whether you want to also filter out soft references would depend a bit on the objects in question – for example, if you are using `SoftReference` with a cache, you might filter out soft references as well to confirm that nothing *other than* your cache is holding onto these objects.

Filtering out weak references (or whatever) brings up another tab containing the GC roots preventing our activity from being garbage collected:


Overview
Histogram
merge_shortest_paths [selection of 'StaticWidgetActivity'] -excludes java.lang.ref.WeakReference:referent

Class Name	Ref. Objects	Shallow Heap	Ref. Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
class com.commonsware.android.tuning.mat.StaticWidgetActivity @	1	8	152	8
doNotDoThisPlease android.widget.TextView @ 0x44edf1d8	1	488	152	856
mContext com.commonsware.android.tuning.mat.StaticWidgetA	1	152	152	2,528

Figure 21. The GC roots holding onto an activity

This is showing that the *class* for our activity has a data member (`doNotDoThisPlease`) that has a `View`, and that in turn is holding onto our activity via an `mContext` data member. Static data members (i.e., data members of class objects) are classic sources of memory leaks in Java. The Retained Heap column on the far right shows how much memory that individual object (and everything it points to) is keeping around – in this case, about 2.5KB.

Identifying What Else is Floating Around

This helps us find where your own objects are being leaked. What happens if you are leaking other things, though?

One possibility is to examine the rest of the Histogram tab, as it will point out the classes (and primitives) that have the most outstanding instances or hold the most aggregate shallow heap. If you applied a regular expression, you can click on the regular expression and delete it to return to the non-filtered roster. The Histogram tends to report a lot of primitives (e.g., `char[]`), and it will take some experience to learn what is standard Android application "noise" and what might represent problems.

Another way to find leaks is to examine the "dominator tree". The term "dominator tree" comes from graph theory – object A "dominates" object B if the only paths to get to B go through A. In MAT, the dominator tree will bubble up those objects whose retained heap – the total memory the object is responsible for, including objects it links to – are high. Or, as MAT describes it, it lists "the biggest objects".

To get to the dominator tree, you can click its link on the Overview tab, or you can click the corresponding toolbar button:

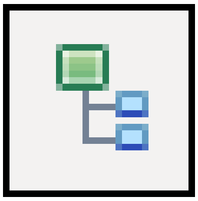
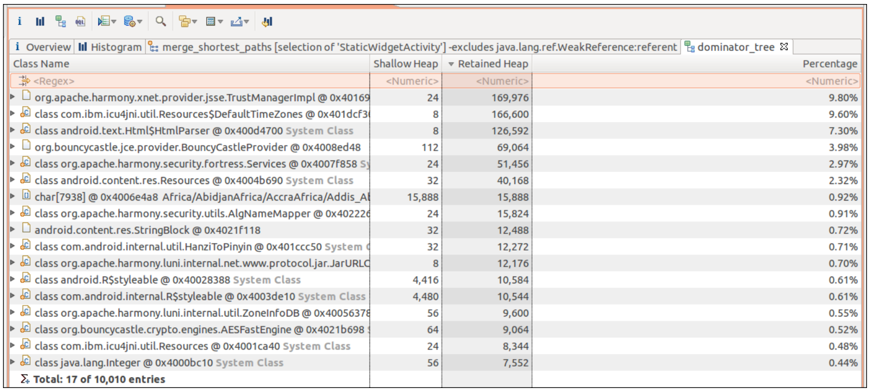


Figure 22. The icon used for the Dominator Tree toolbar button

This will open up another tab in the same tool, showing "the biggest objects" by retained heap:



Overview | Histogram | merge_shortest_paths [selection of 'StaticWidgetActivity'] -excludes java.lang.ref.WeakReference:referent | dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
org.apache.harmony.xnet.provider.jsse.TrustManagerImpl @ 0x40169	24	169,976	9.80%
class com.ibm.icu4jni.util.Resources$DefaultTimeZones @ 0x401dcf3	8	166,600	9.60%
class android.text.Html$HtmlParser @ 0x400d4700 System Class	8	126,592	7.30%
org.bouncycastle.jce.provider.BouncyCastleProvider @ 0x4008ed48	112	69,064	3.98%
class org.apache.harmony.security.fortress.Services @ 0x4007f858 Sy	24	51,456	2.97%
class android.content.res.Resources @ 0x4004b690 System Class	32	40,168	2.32%
char[7938] @ 0x4006e4a8 Africa/AbidjanAfrica/AccraAfrica/Addis_Al	15,888	15,888	0.92%
class org.apache.harmony.security.utils.AlgNameMapper @ 0x40222	24	15,824	0.91%
android.content.res.StringBlock @ 0x4021f118	32	12,488	0.72%
class com.android.internal.util.HanziToPinyin @ 0x401ccc50 System	32	12,272	0.71%
class org.apache.harmony.luni.internal.net.www.protocol.jar.JarURLC	8	12,176	0.70%
class android.R$styleable @ 0x40028388 System Class	4,416	10,584	0.61%
class com.android.internal.R$styleable @ 0x4003de10 System Class	4,480	10,544	0.61%
class org.apache.harmony.luni.internal.util.ZoneInfoDB @ 0x40056378	56	9,600	0.55%
class org.bouncycastle.crypto.engines.AESFastEngine @ 0x4021b698	64	9,064	0.52%
class com.ibm.icu4jni.util.Resources @ 0x4001ca40 System Class	24	8,344	0.48%
class java.lang.Integer @ 0x4000bc10 System Class	56	7,552	0.44%
Total: 17 of 10,010 entries			

Figure 23. The MAT Dominator Tree tab

You can display more by right-clicking over the Total row at the bottom and choosing "Next 25".

Here too, the roster will mostly be system objects (e.g., `org.bouncycastle` for the `javax.crypto` implementation). What you would be looking for are objects that you might be interacting with more directly that perhaps you are leaking, such as a Bitmap.

If you find something of interest, right-clicking over the object and choosing "Path to GC Roots" or "Merge Shortest Paths to GC Roots" will help you track down what is holding onto the object, akin to the similar feature in the Histogram.

Some Leaks and Their MAT Analysis

Let's now take a look at some common leak scenarios in Android and see how we find out whether we have a leak and what is causing it. All of the projects demonstrated below are in the `MAT` directory of the book's source code.

Widget in Static Data Member

The screen shots from above are mostly taken from the `MAT/StaticWidget` sample project, where we do something naughty:

```
package com.commonsware.android.tuning.mat;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;

public class StaticWidgetActivity extends Activity {
  @SuppressWarnings("unused")
  static private View doNotDoThisPlease;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    doNotDoThisPlease=findViewById(R.id.make_me_static);
  }
}
```

We take a widget (specifically the auto-generated `TextView`) and put it in a static data member, and never replace it with null.

As a result, even if the user presses BACK to get out of the activity, the static data member holds onto `TextView`, which itself has a reference back to our Activity.

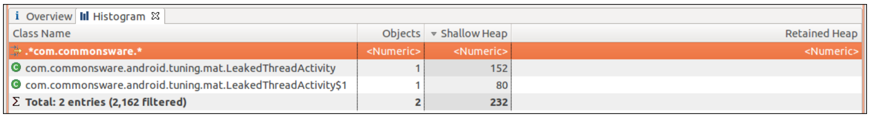
Usually, you will pick this sort of leak up by scanning on your own application's package, as your activity will appear in there. If you are using multiple packages in your application (e.g., yours and a third-party activity), you might need to also check the third-party package to see if any of its objects are being leaked. Whether those leaks are the fault of your code or the third party's own code will vary, of course.

Leaked Thread

You can see similar results when you leak a thread, such as in the MAT/LeakedThread sample project:

```java
package com.commonsware.android.tuning.mat;

import android.app.Activity;
import android.os.Bundle;
import android.os.SystemClock;

public class LeakedThreadActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    new Thread() {
      public void run() {
        while(true) {
          SystemClock.sleep(100);
        }
      }
    }.start();
  }
}
```

Here, if we filter on `com.commonsware` in the Histogram, we see two entries:



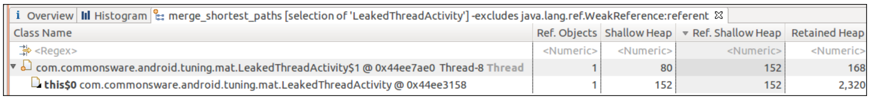
Overview | Histogram

Class Name	Objects	Shallow Heap	Retained Heap
.*com.commonsware.*	<Numeric>	<Numeric>	<Numeric>
com.commonsware.android.tuning.mat.LeakedThreadActivity	1	152	
com.commonsware.android.tuning.mat.LeakedThreadActivity$1	1	80	
Σ Total: 2 entries (2,162 filtered)	2	232	

Figure 24. The LeakedThreadActivity Histogram

As with other places in Java (e.g., stack traces), the `$` syntax in a class name refers to an inner class, and `$1` refers to the first anonymous inner class.

If we look at the GC roots for the activity, we see:



Overview | Histogram | merge_shortest_paths [selection of 'LeakedThreadActivity'] -excludes java.lang.ref.WeakReference:referent

Class Name	Ref. Objects	Shallow Heap	Ref. Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>	<Numeric>
com.commonsware.android.tuning.mat.LeakedThreadActivity$1 @ 0x44ee7ae0 Thread-8 Thread	1	80	152	168
this$0 com.commonsware.android.tuning.mat.LeakedThreadActivity @ 0x44ee3158	1	152	152	2,320

Figure 25. The GC roots for LeakedThreadActivity

The root is a thread, as denoted by the "Thread" annotation on the end of the root entry. We see that the `Thread` object itself is our `$1` inner class instance, and it holds onto the activity via the implicit reference every non-static inner class has to its outer class instance (`this$0`).

Any running thread will cause anything it can reach to remain in the heap and not get garbage collected. An inner class implementation of the `Thread` – which most code examples will use, in one form or fashion – will leak the outer class instance. Hence, the lessons to be learned here are:

- Leaking threads leaks memory
- Consider using static inner classes, or separate classes, rather than non-static inner classes, so you do not cause objects to be held onto unnecessarily and unexpectedly

All Sorts of Bugs

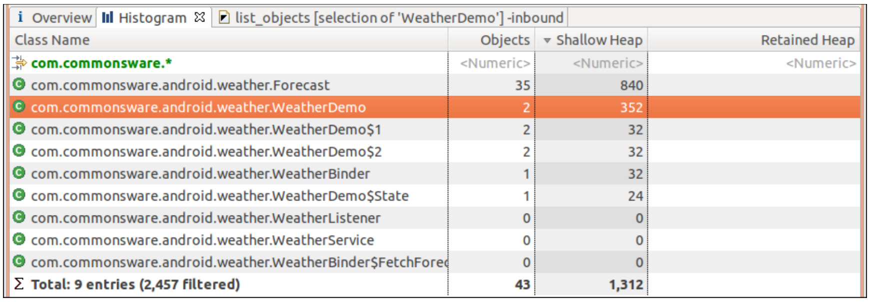
Let's now example `MAT/RandomAppOfCrap`. This is a variation on an example from The Busy Coder's Guide to Android Development, showing using a bound service that connects to a Web service – in this case, the US National Weather Service. In this modified version, a number of leak-related bugs were introduced.

Leaks Via Configuration Changes

The `WeatherDemo` activity implements `onRetainNonConfigurationInstance()`, returning a `State` object. `State` is an inner class of `WeatherDemo`, but not a static inner class.

This is not a good idea.

When you search the Histogram for `com.commonsware` after loading a weather forecast (e.g., run the app and use DDMS to push over a location fix) and rotating the screen, you see that there are two instances of `WeatherDemo` floating around the heap:

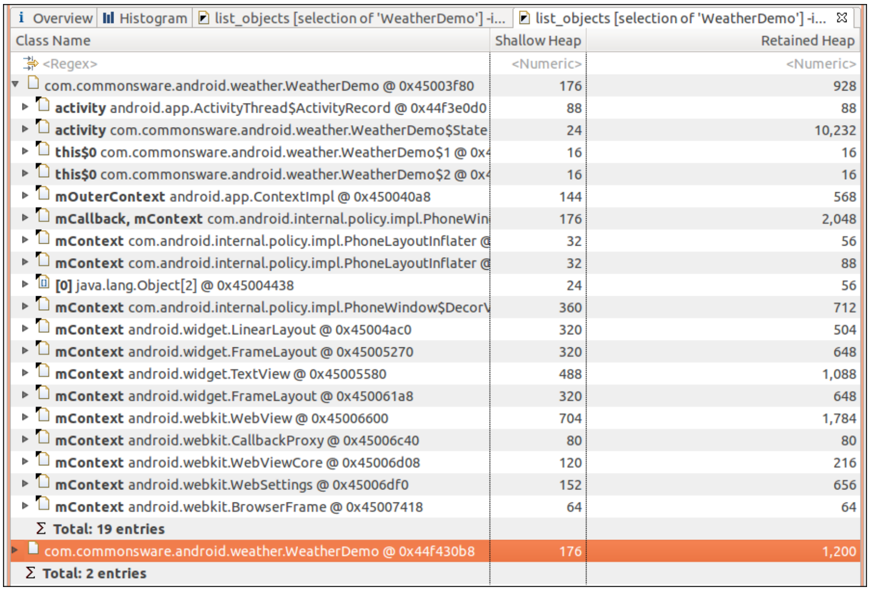


i Overview | Histogram ☒ | list_objects [selection of 'WeatherDemo'] -inbound

Class Name	Objects	Shallow Heap	Retained Heap
com.commonsware.*	<Numeric>	<Numeric>	<Numeric>
com.commonsware.android.weather.Forecast	35	840	
com.commonsware.android.weather.WeatherDemo	2	352	
com.commonsware.android.weather.WeatherDemo$1	2	32	
com.commonsware.android.weather.WeatherDemo$2	2	32	
com.commonsware.android.weather.WeatherBinder	1	32	
com.commonsware.android.weather.WeatherDemo$State	1	24	
com.commonsware.android.weather.WeatherListener	0	0	
com.commonsware.android.weather.WeatherService	0	0	
com.commonsware.android.weather.WeatherBinder$FetchFore	0	0	
Σ **Total: 9 entries (2,457 filtered)**	**43**	**1,312**	

Figure 26. The com.commonsware objects in the RandomAppOfCrap heap

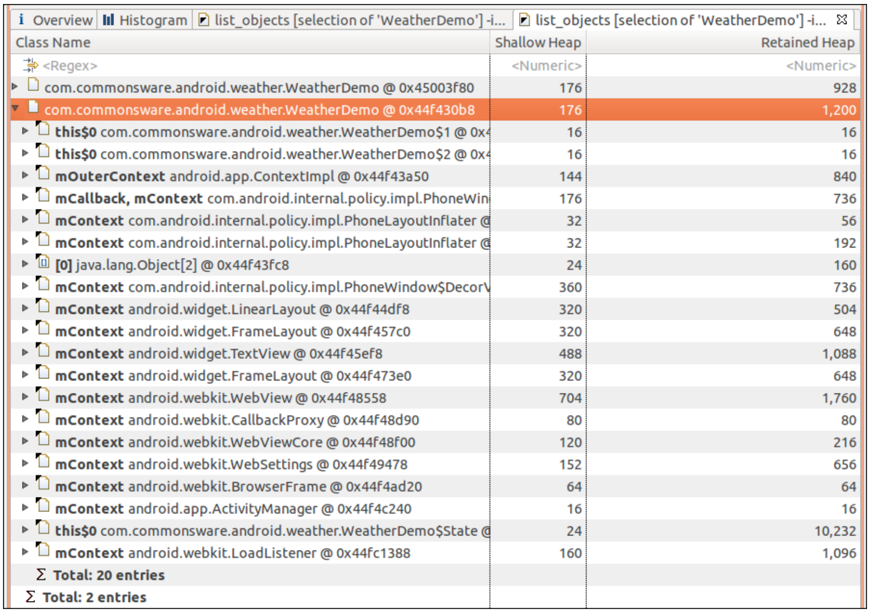
To figure out what those objects are, you can right-click over a class in the Histogram and choose "List Objects" from the context menu. The fly-out sub-menu will let you choose to show incoming references (who points to these objects) or outgoing references (what these objects point to). In this case, showing incoming references will bring up the following:



i Overview | Histogram | list_objects [selection of 'WeatherDemo'] -i... | list_objects [selection of 'WeatherDemo'] -i... ☒

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.commonsware.android.weather.WeatherDemo @ 0x45003f80	176	928
activity android.app.ActivityThread$ActivityRecord @ 0x44f3e0d0	88	88
activity com.commonsware.android.weather.WeatherDemo$State	24	10,232
this$0 com.commonsware.android.weather.WeatherDemo$1 @ 0x4	16	16
this$0 com.commonsware.android.weather.WeatherDemo$2 @ 0x4	16	16
mOuterContext android.app.ContextImpl @ 0x450040a8	144	568
mCallback, mContext com.android.internal.policy.impl.PhoneWin	176	2,048
mContext com.android.internal.policy.impl.PhoneLayoutInflater @	32	56
mContext com.android.internal.policy.impl.PhoneLayoutInflater @	32	88
[0] java.lang.Object[2] @ 0x45004438	24	56
mContext com.android.internal.policy.impl.PhoneWindow$DecorV	360	712
mContext android.widget.LinearLayout @ 0x450044c0	320	504
mContext android.widget.FrameLayout @ 0x45005270	320	648
mContext android.widget.TextView @ 0x45005580	488	1,088
mContext android.widget.FrameLayout @ 0x450061a8	320	648
mContext android.webkit.WebView @ 0x45006600	704	1,784
mContext android.webkit.CallbackProxy @ 0x45006c40	80	80
mContext android.webkit.WebViewCore @ 0x45006d08	120	216
mContext android.webkit.WebSettings @ 0x45006df0	152	656
mContext android.webkit.BrowserFrame @ 0x45007418	64	64
Σ **Total: 19 entries**		
com.commonsware.android.weather.WeatherDemo @ 0x44f430b8	176	1,200
Σ **Total: 2 entries**		

Figure 27. The incoming references for one WeatherDemo instance


Overview
Histogram
list_objects [selection of 'WeatherDemo'] -i...
list_objects [selection of 'WeatherDemo'] -i...

Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
com.commonsware.android.weather.WeatherDemo @ 0x45003f80	176	928
com.commonsware.android.weather.WeatherDemo @ 0x44f430b8	176	1,200
this$0 com.commonsware.android.weather.WeatherDemo$1 @ 0x4	16	16
this$0 com.commonsware.android.weather.WeatherDemo$2 @ 0x4	16	16
mOuterContext android.app.ContextImpl @ 0x44f43a50	144	840
mCallback, mContext com.android.internal.policy.impl.PhoneWin	176	736
mContext com.android.internal.policy.impl.PhoneLayoutInflater @	32	56
mContext com.android.internal.policy.impl.PhoneLayoutInflater @	32	192
[0] java.lang.Object[2] @ 0x44f43fc8	24	160
mContext com.android.internal.policy.impl.PhoneWindow$DecorV	360	736
mContext android.widget.LinearLayout @ 0x44f44df8	320	504
mContext android.widget.FrameLayout @ 0x44f457c0	320	648
mContext android.widget.TextView @ 0x44f45ef8	488	1,088
mContext android.widget.FrameLayout @ 0x44f473e0	320	648
mContext android.webkit.WebView @ 0x44f48558	704	1,760
mContext android.webkit.CallbackProxy @ 0x44f48d90	80	80
mContext android.webkit.WebViewCore @ 0x44f48f00	120	216
mContext android.webkit.WebSettings @ 0x44f49478	152	656
mContext android.webkit.BrowserFrame @ 0x44f4ad20	64	64
mContext android.app.ActivityManager @ 0x44f4c240	16	16
this$0 com.commonsware.android.weather.WeatherDemo$State @	24	10,232
mContext android.webkit.LoadListener @ 0x44fc1388	160	1,096
Σ **Total: 20 entries**		
Σ **Total: 2 entries**		

Figure 28. The incoming references for the other WeatherDemo instance

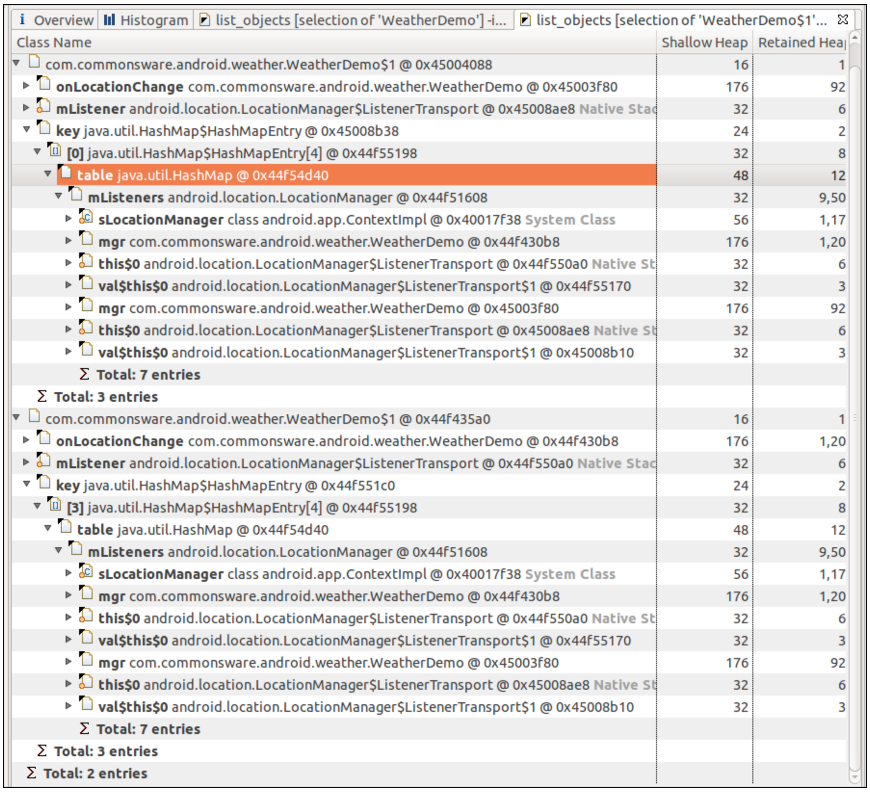
The eight-digit hex numbers shown after the `@` sign are the object identifiers for each of the referred-to objects. You can use this to distinguish which objects are the same.

What you will notice is that both `WeatherDemo` instances are pointed to by the `State` object. In one, it is referred to by the `activity` data member. In the other, it is referred to by `this$0` – the implicit reference an inner class instance has on the outer class instance. Since both `WeatherDemo` instances hold onto the `State` via the `state` data member, this means that one `WeatherDemo` instance (the foreground one) is holding an indirect reference, via the `State`, to the other now-destroyed `WeatherDemo` instance. This is a leak.

The solution for this would be to use a static inner class for `State`, eliminating the implicit reference and breaking this connection.

Leaks from Unregistered System Listeners

We also see from our filtered Histogram that we have two retained instances of the `$1` inner class. Displaying incoming references to those objects shows us that those are the `LocationListener` objects we are using to get our GPS fixes:



```
i Overview | Histogram | list_objects [selection of 'WeatherDemo'] -i... | list_objects [selection of 'WeatherDemo$1'...
Class Name                                                                                          Shallow Heap  Retained Hea
▼ com.commonsware.android.weather.WeatherDemo$1 @ 0x45004088                                                  16             1
  ▶ onLocationChange com.commonsware.android.weather.WeatherDemo @ 0x45003f80                                176            92
  ▶ mListener android.location.LocationManager$ListenerTransport @ 0x45008ae8 Native Stac                     32             6
  ▼ key java.util.HashMap$HashMapEntry @ 0x45008b38                                                           24             2
    ▼ [0] java.util.HashMap$HashMapEntry[4] @ 0x44f55198                                                      32             8
      ▼ table java.util.HashMap @ 0x44f54d40                                                                  48            12
        ▼ mListeners android.location.LocationManager @ 0x44f51608                                            32          9,50
          ▶ sLocationManager class android.app.ContextImpl @ 0x40017f38 System Class                          56          1,17
          ▶ mgr com.commonsware.android.weather.WeatherDemo @ 0x44f430b8                                     176          1,20
          ▶ this$0 android.location.LocationManager$ListenerTransport @ 0x44f550a0 Native St                   32             6
          ▶ val$this$0 android.location.LocationManager$ListenerTransport$1 @ 0x44f55170                      32             3
          ▶ mgr com.commonsware.android.weather.WeatherDemo @ 0x45003f80                                     176            92
          ▶ this$0 android.location.LocationManager$ListenerTransport @ 0x45008ae8 Native St                   32             6
          ▶ val$this$0 android.location.LocationManager$ListenerTransport$1 @ 0x45008b10                      32             3
          Σ Total: 7 entries
    Σ Total: 3 entries
▼ com.commonsware.android.weather.WeatherDemo$1 @ 0x44f435a0                                                  16             1
  ▶ onLocationChange com.commonsware.android.weather.WeatherDemo @ 0x44f430b8                                176          1,20
  ▶ mListener android.location.LocationManager$ListenerTransport @ 0x44f550a0 Native Stac                     32             6
  ▼ key java.util.HashMap$HashMapEntry @ 0x44f551c0                                                           24             2
    ▼ [3] java.util.HashMap$HashMapEntry[4] @ 0x44f55198                                                      32             8
      ▼ table java.util.HashMap @ 0x44f54d40                                                                  48            12
        ▼ mListeners android.location.LocationManager @ 0x44f51608                                            32          9,50
          ▶ sLocationManager class android.app.ContextImpl @ 0x40017f38 System Class                          56          1,17
          ▶ mgr com.commonsware.android.weather.WeatherDemo @ 0x44f430b8                                     176          1,20
          ▶ this$0 android.location.LocationManager$ListenerTransport @ 0x44f550a0 Native St                   32             6
          ▶ val$this$0 android.location.LocationManager$ListenerTransport$1 @ 0x44f55170                      32             3
          ▶ mgr com.commonsware.android.weather.WeatherDemo @ 0x45003f80                                     176            92
          ▶ this$0 android.location.LocationManager$ListenerTransport @ 0x45008ae8 Native St                   32             6
          ▶ val$this$0 android.location.LocationManager$ListenerTransport$1 @ 0x45008b10                      32             3
          Σ Total: 7 entries
    Σ Total: 3 entries
Σ Total: 2 entries
```

Figure 29. The incoming references for the WeatherDemo$1 instances

Tracing through the incoming references, we see that the `ContextImpl` class holds a static reference to the `LocationManager` system service in our process, and `LocationManager` has an `mListeners` data member which is a list of all registered `LocationListener` instances.

Alas, in `WeatherDemo`, we are registering a `LocationListener` and never unregistering it. Since our `LocationListener` is an inner class, not only is the

`LocationListener` itself leaked, but it prevents our destroyed `WeatherDemo` object from being garbage collected.

This same pattern can be seen for many of the system services – if you register a listener, you must ensure that you unregister it to prevent leaks.

What MAT Won't Tell You

MAT is not a universal solution. It may not tell you of all possible leaks.

For example, if you bind to a service, the `ServiceConnection` object you create is held onto, indirectly, by the OS itself. That is how you can use the `ServiceConnection` to unbind from the service later on. However, if you examine MAT, you will see no evidence of this, as MAT is limited to examining *your own* process and cannot report about references that are triggered by other processes.

MAT also will not report anything that is part of the native heap (i.e., what you get with a C `malloc()` call) – it only reports on the Dalvik heap. Hence, MAT will not reflect the actual memory consumption of bitmap images on Gingerbread and earlier environments. You may wish to do some testing of your app on Honeycomb, not just for any tablet support you may offer, but to get more complete results from MAT.

PART IV – Battery Performance

CHAPTER 11

Issues with Battery Life

Most Android devices are powered by batteries – Google TV is the biggest class of device that is not. Batteries are wonderful gizmos with one major problem: they are *always* running out of power.

Hence, users are very sensitive to battery consumption. Their ability to use their phones as actual *phones*, let alone for Android apps, depends on having enough battery power. The more apps drain the battery, the more frequently the user has to find a way to recharge the phone, and the more frequently the user fails and their phone shuts down.

The catch is that you may not notice the battery issues in your day-to-day development. The Android emulator's emulated battery does not drain based on you running your app. Your devices are often connected to your development machine via USB for testing and debugging, meaning they are perpetually being charged. Unless you are a regular user of your own app, you might not notice any increased power drain.

This part of the book is focused on helping you understand what is draining power and what you can do to be kinder and gentler on your users' batteries.

You're Getting Blamed

Users, for better or worse, have limited ability to determine what is responsible for draining the battery of their phone. Their #1 tool for this is the "Power Usage Summary" screen in the Settings app, sometimes referred to as the "battery blame screen".

<< screenshot >>

This lists both device features (e.g., the display) and applications. Android incrementally improves the accuracy of this screen with each passing release, trying to make sure the user understands what specifically is consuming the power.

If your application starts appearing on this screen, and the user does not feel that it is justified, the user is likely to become irritated with you.

Now, your appearance on this list might be perfectly reasonable. If you have written a video player app, and the user has just watched a few hours' worth of video, it is very likely that you will appear on this list and will be justified in your battery consumption.

However, anything that you can do to not appear on this screen, or appear lower in the list, will help with user acceptance of your app.

This part of the book will show you how to measure your power usage and ways of trying to use less of it.

Stretching Out the Last mWh

Sometimes, what the user wants your app to do in one case is not what the user wants your app to do in other cases. Serious power-draining might be reserved for when the device is plugged in, or when the device has at least such-and-so power remaining. The user may value the last milliwatt-hours (mWh) more than others and want your application to use less power in those circumstances.

Hence, if your application polls the Internet, you might offer a feature to poll less frequently, or perhaps not at all, when power is low. If your application uses GPS to find a location (e.g., automatic "check-ins" to social networks like Foursquare), you might offer to skip such actions when the battery is low. You might want to signal to the user when the battery gets low during playback of a video, or during the game they are in. And so on.

This part of the book will help you identify when the battery is low and strategies for making use of that information.

CHAPTER 12

Focus On: MDP and Trepn

You can measure power drain in one of two ways:

- Rip open a device enough to hook up a multi-tester to the proper leads to measure physically on the device how much power is being drained from the battery. You will need to either get a very sophisticated recording multi-tester, or perhaps cross-train a court stenographer to be able to record the power levels consumed as fast as possible.
- You find a device that can do this sort of recording automatically.

Since recording multi-testers and court stenographers are expensive, you might head in the latter direction. Fortunately, Qualcomm makes a series of devices – the Mobile Development Platform, or MDP – that can record real-time power consumption. Qualcomm also makes a tool that can interpret this information, called Trepn.

In this chapter, we will examine the MDP and Trepn in greater detail, so you can determine what sorts of information this device can give you.

What Are You Talking About?

It is very likely that even seasoned Android developers will have never heard of MDP or Trepn. You will not find an MDP in your local electronics store. You will not even find them on eBay (most of the time). And since

Trepn is largely useless without access to an MDP's power recordings, only those who have run across an MDP are likely to have also heard of Trepn.

Of course, since you are reading this book, it is clear that you are an exemplary Android developer, one thirsting for knowledge and who therefore might be interested in learning more about these hidden gems.

What's an MDP?

The Qualcomm MDP is a mobile phone, but not one designed for consumer use. Rather, it is a reference platform for a Qualcomm mobile CPU. There are two MDP models, one each for two Qualcomm processors: the MSM8660 and the MSM8655.

As a reference platform, this device is not necessarily designed to be regularly used. Instead, it is designed to show off a number of advanced hardware capabilities and allow developers to test on them. For example, the MDP for the MSM8660 has:

- Dual cores (1.5GHz each)
- 1080p video recording and playback
- 3D output via HDMI
- a 13 megapixel main camera

These are all at or above most mainstream devices, as of the device's release in late spring 2011.

The MDP also has additional instrumentation designed to assist with testing applications, and that is where Trepn comes in.

What's a Trepn?

The MDP has specialized hooks in the firmware to monitor power consumption by various components: CPU, radios, display, etc. Trepn is an application, built into the MDP, that can collect, record, and display that data and related information. Developers can use Trepn to determine how much power their application uses while it runs through a test suite, for example.

Trepn runs on the device itself, though it does save its results as CSV files for possible offline analysis. Trepn, therefore, is not something that you run on your development machine or in your Web browser, but on the MDP itself.

The Big Problem: Cost

The MDP MSM8660, at the time of this writing, runs nearly $1,400. This means that few Android application developers will have direct hands-on access to the MDP.

A previous version of the MDP also had issues with the purchase agreement you had to abide by when obtaining an MDP from BSQUARE (Qualcomm's retailer/front-line support firm for the MDP). For example, the purchase agreement would have forbidden this chapter from being published, as it includes results from running tests on the MDP. However, the purchase agreement for the MSM8660 MDP is more reasonable.

Running Trepn Tests

Measuring your power consumption using Trepn is fairly straightforward, particularly for simple cases… with one big limitation.

First, you will need to get your app on the MDP. You may even wish to run the app once on the MDP – if your concerns are power consumption over the long haul, getting all of your app initialization logic done before you start measuring power is probably a good move.

Next, run the "Trepn Profiler" application on the MDP, found in the launcher like any other activity.

<< screenshot >>

Then, click the "Begin Profiling" button. This will bring up a dialog box where you can select an application on the MDP that Trepn should launch and monitor.

<< screenshot >>

Note that this means you cannot readily use Trepn to measure the power consumed by a unit test suite or other form of instrumentation. You may wish to organize your code into an Android library project with a separate project for the UI front end, with additional projects for testing various power consumption scenarios that you use with Trepn.

Once you choose an application and click the Start button in the dialog, Trepn will gather a few seconds of "warmup" data, then run your app. You are welcome to interact with your application at this point, if your app is interactive and you have not otherwise automated the testing. When you are done, return to the Trepn Profiler activity (e.g., through the Notification in the status bar) and click the "Stop Profiling" button. You will be prompted for the name of a directory in which to save the data.

And that's it!

Recording Application States

The problem is, Trepn does not intrinsically know much about your application. It is simply recording power usage while your application is running. Trepn has no way of knowing when certain features of your app are used or certain calculations are run.

Unless you tell it.

You can send a broadcast Intent that Trepn will pick up, indicating what "application state" your app is now in. Here, an "application state" is simply some integer – it will be up to you to map integers to various portions of your application logic (e.g., 1 is normal, 2 is during your data download, 3 is during your data export process). If you tell Trepn the states of your application, it will not only record the overall results but the "splits" for each one of your states.

For example, the `Power/Downloader` sample application is a slightly modified version of one from *The Busy Coder's Guide to Android Development*. In that book, the sample app had a large button – clicking the button would kick off a download of a ~700KB PDF file in a background thread via an `IntentService`. This book's version of the sample skips the button – launching the activity will introduce a five-second pause, then the download will begin automatically. The activity will be finished once the download is complete.

Along the way, we let Trepn know when the download work begins:

```
@Override
public void onHandleIntent(Intent i) {
 Intent trepn=new Intent("com.quicinc.Trepn.UpdateAppState");

 trepn.putExtra("com.quicinc.Trepn.UpdateAppState.Value",
 1337);
 sendBroadcast(trepn);
```

You need to create an `Intent` for the `com.quicinc.Trepn.UpdateAppState` action, add an extra with your integer keyed as `com.quicinc.Trepn.UpdateAppState.Value`, then send the broadcast.

The fact that Trepn uses broadcasts here means you will want your application states to be fairly coarse-grained. You cannot realistically update the state more than once every couple of seconds, and the asynchronous nature of sending broadcasts means that your work might begin before the state itself is recorded.

Examining Trepn Results

You have two ways to look at the data that Trepn collects. There is an on-device UI, integrated as part of the Trepn application. Or, you can grab the raw data and perform your own offline analysis using your choice of tools.

On-Device

Either before stopping profiling, or by reloading the Trepn session via the "View Saved Sessions" button, you can view a graph of power consumption:

<< screenshot >>

or click "View Stats" for a tabular rendition of the data:

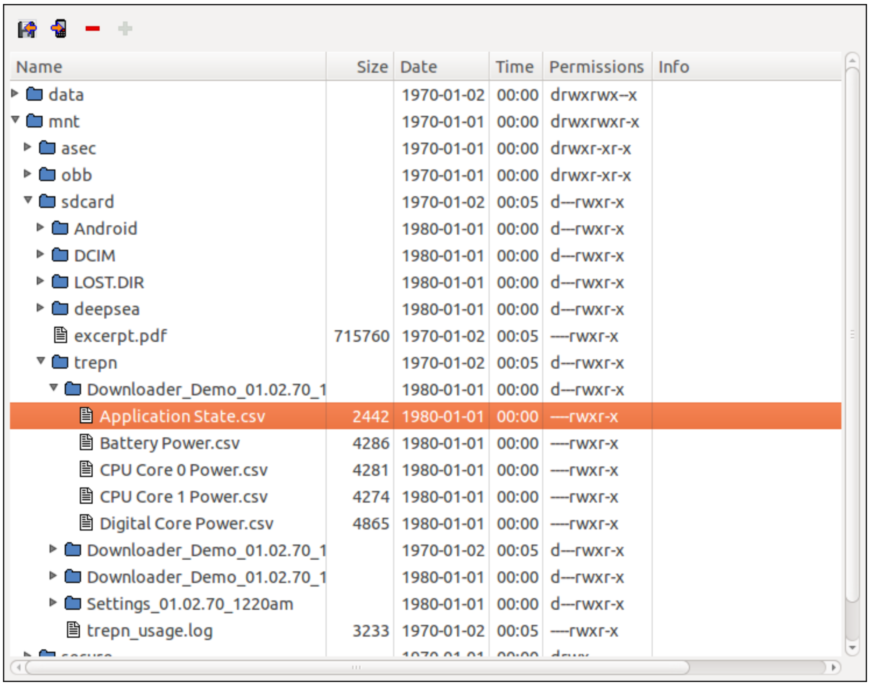
<< screenshot >>

By default, Trepn will record a handful of values, such as the power consumed overall and by the two CPU cores. In the Trepn settings activity, though, you can toggle on or off any number of other values to record, plus indicate if they should be displayed in the resulting graph.

Both the graph and the table will show your application states. On the graph, the changes in your application state value will be graphed along with everything else. The table will show how much time and power was consumed in each of your states – probably a more valuable means of interpreting the results.

Off-Device

If you browse the external storage of the MDP, you will find a trepn directory that contains the saved sessions from your tests:



Name	Size	Date	Time	Permissions	Info
data		1970-01-02	00:00	drwxrwx–x	
mnt		1970-01-01	00:00	drwxrwxr-x	
asec		1970-01-01	00:00	drwxr-xr-x	
obb		1970-01-01	00:00	drwxr-xr-x	
sdcard		1970-01-02	00:05	d—rwxr-x	
Android		1980-01-01	00:00	d—rwxr-x	
DCIM		1980-01-01	00:00	d—rwxr-x	
LOST.DIR		1980-01-01	00:00	d—rwxr-x	
deepsea		1980-01-01	00:00	d—rwxr-x	
excerpt.pdf	715760	1970-01-02	00:05	—-rwxr-x	
trepn		1970-01-02	00:05	d—rwxr-x	
Downloader_Demo_01.02.70_1		1980-01-01	00:00	d—rwxr-x	
Application State.csv	2442	1980-01-01	00:00	—-rwxr-x	
Battery Power.csv	4286	1980-01-01	00:00	—-rwxr-x	
CPU Core 0 Power.csv	4281	1980-01-01	00:00	—-rwxr-x	
CPU Core 1 Power.csv	4274	1980-01-01	00:00	—-rwxr-x	
Digital Core Power.csv	4865	1980-01-01	00:00	—-rwxr-x	
Downloader_Demo_01.02.70_1		1970-01-02	00:05	d—rwxr-x	
Downloader_Demo_01.02.70_1		1980-01-01	00:00	d—rwxr-x	
Settings_01.02.70_1220am		1980-01-01	00:00	d—rwxr-x	
trepn_usage.log	3233	1970-01-02	00:05	—-rwxr-x	

Figure 30. External storage on an MDP, showing saved sessions

For each collected statistic for each saved session, there will be a CSV file containing the raw data. The columns for the CSV file will vary by statistic, though all will have a time offset column to indicate when the value was recorded.

For example, here is an extract from the `Battery Power.csv` file from one Trepn run with ellipses added to show where portions of the file were removed for brevity:

```
Time (ms),Battery Power (uA),Battery Power (uW)
-4914,2600,10784
-4814,2600,10784
-4714,2600,10784
...
-104,2600,10784
-2,2600,10784
99,2600,10784
198,2600,10784
299,2600,10784
...
```

```
5024,2600,10784
5125,2600,10784
5224,2600,10784
5325,2600,10784
5427,2600,10784
5525,2600,10784
5630,2600,10784
5732,2600,10784
5833,2600,10784
5931,39800,165090
6031,3000,12443
6134,2600,10784
6234,2600,10784
...
```

Time values less than 0 represent the "warmup" period before Trepn actually runs your application. In this case, the battery power is shown both in micro-amps (uA) and micro-watts (uW).

To correlate these events with your application states, you will also need to examine the `Application State.csv` file:

```
Time (ms),Application State
-4950,0
-4849,0
-4749,0
...
-134,0
-34,0
66,0
167,0
267,0
367,0
...
5092,0
5136,1337
5193,1337
5293,1337
5393,1337
5494,1337
5594,1337
5694,1337
5794,1337
5895,1337
5996,1337
6096,1337
6196,1337
6297,1337
...
```

The time offsets will not line up precisely (for whatever reason), but will show the saved application state value at the specific offsets. So, between 5.092 and 5.136 seconds after the actual test began, our application state shifted from the default to 1337, corresponding to the value we sent over in the broadcast `Intent` extra. All power levels after that point would be related to the download operation.

In principle, one could import these into a spreadsheet or craft tools to parse the CSV data and create other visual representations, particularly in ways that can be used without the MDP being around.

CHAPTER 13

Other Power Measurement Options

Given the sheer expense of the Qualcomm MDP, few developers will have direct access to one, despite the detailed power statistics one can glean from Trepn. There are free alternatives, but they all have substantial limits when compared to the combination of the MDP and Trepn.

PowerTutor

Perhaps the best-known third-party power analyzer is PowerTutor. PowerTutor is the outcome of a research project from the University of Michigan, with a bit of assistance from Google. In principle, PowerTutor is capable of letting you know power consumption on a device, much along the lines of what Trepn can record on a Qualcomm MDP. In practice, PowerTutor is significantly less powerful and sophisticated.

PowerTutor was created with the HTC Dream (T-Mobile G1), HTC Magic (T-Mobile G2), and Nexus One in mind. Its power output values will be as accurate as they could make it for those devices. If you run PowerTutor on other hardware, the results will be less accurate.

You can obtain PowerTutor from the Android Market, or from the PowerTutor Web site, or you can compile it from source.

PowerTutor is not tied to testing a particular application. As such, you can simply run PowerTutor whenever you want from its launcher icon, then press "Start Power Profiler" in the main activity:

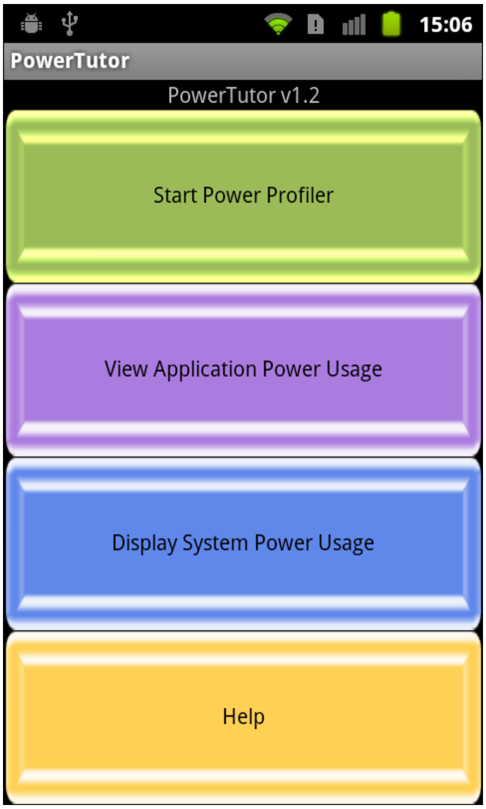
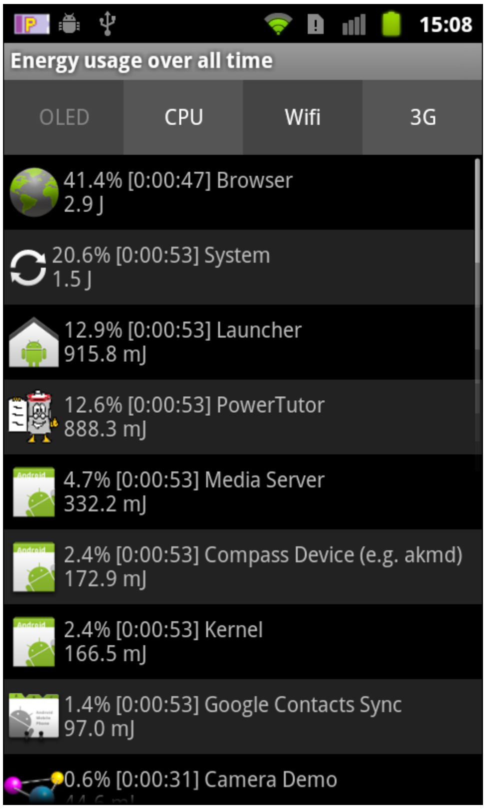


Figure 31. The PowerTutor main activity

At this point, you can start playing with your application, or running your unit test suite, or whatever. When you want to get an idea of how much power you have been consuming, you can switch back to the PowerTutor activity and choose "View Application Power Usage". This brings up a list of processes and toggle buttons to show various power consumption values for each:



41.4% [0:00:47] Browser
2.9 J
20.6% [0:00:53] System
1.5 J
12.9% [0:00:53] Launcher
915.8 mJ
12.6% [0:00:53] PowerTutor
888.3 mJ
4.7% [0:00:53] Media Server
332.2 mJ
2.4% [0:00:53] Compass Device (e.g. akmd)
172.9 mJ
2.4% [0:00:53] Kernel
166.5 mJ
1.4% [0:00:53] Google Contacts Sync
97.0 mJ
0.6% [0:00:31] Camera Demo

Figure 32. The PowerTutor application roster

Tapping the list entry brings up a graph for that particular process, though since this information is only available while PowerTutor is recording new data, the graph is usually empty unless you have logic running in the background:

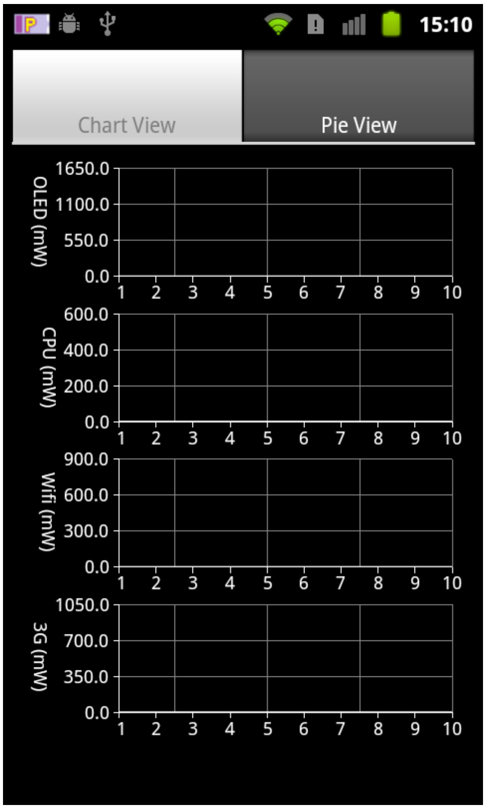


Figure 33. The PowerTutor live charts for a single process current power consumption

You can also bring up a charts showing what portion of your power consumption came from various sources for the whole device, such as a pie chart of current consumption:

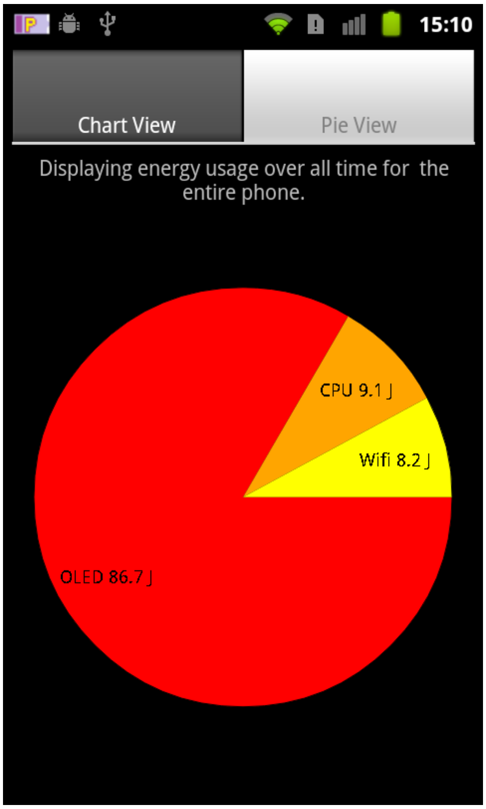


Figure 34. The PowerTutor pie chart for current overall power consumption

Given that the source code is available, one might augment PowerTutor to:

- Saving results, both as data files for offline analysis (akin to Trepn's CSV files) or for viewing charts and tables on the device when data is not being actively collected
- Allowing one to record application states, akin to Trepn, to better correlate application functionality to saved power results

Battery Screen in Settings Application

Of course, what developers tend to focus on most with power is the battery consumption screen in the Settings application:

<< screenshot >>

After all, this is what users will tend to focus on – anything showing up in here is a source of blame for whatever power woes the user believes she is experiencing. Conversely, if your application does not show up in this screen during normal operation, then there is no compelling reason for you to do further analysis, as users will tend to be oblivious to your actual power consumption.

If you do show up in the list, tapping on your entry can give you some more details of what power you consumed and why:

<< screenshot >>

BatteryInfo Dump

Yet another possibility is to use the `adb shell dumpsys batteryinfo` command from your command prompt or terminal on your development workstation. This will emit a fair amount of data that probably means something to somebody, such as general device information:

```
Battery History:
 -1h00m56s463ms 096 20030002 status=discharging health=good plug=none temp=191
volt=4060 +screen +wake_lock +sensor brightness=medium
 -1h00m52s490ms 096 22030302 +wifi phone_state=off
 -1h00m51s844ms 096 2703d102 +phone_scanning +wifi_running phone_state=out
data_conn=other
 -1h00m49s303ms 096 2743d102 +wifi_scan_lock
 -57m48s766ms 095 2743d102
 -53m24s627ms 095 2743d100 brightness=dark
 -53m17s620ms 095 0741d100 -screen -wake_lock
 -53m17s107ms 095 0740d100 -sensor
 -38m17s007ms 095 0642d100 -wifi_running +wake_lock
 -38m08s998ms 095 0640d100 -wake_lock
 -54s781ms 095 4640d100 status=full plug=usb temp=193 volt=4084 +plugged

Per-PID Stats:
 PID 96 wake time: +12s75ms
 PID 177 wake time: +1s13ms
 PID 458 wake time: +1s898ms
 PID 326 wake time: +3s925ms
 PID 205 wake time: +2s107ms
 PID 415 wake time: +843ms
 PID 96 wake time: +281ms

Statistics since last charge:
 System starts: 0, currently on battery: false
 Time on battery: 1h 0m 1s 682ms (0.3%) realtime, 8m 21s 883ms (0.0%) uptime
 Total run time: 16d 11h 13m 34s 654ms realtime, 2h 9m 37s 404ms uptime,
 Screen on: 7m 37s 868ms (12.7%), Input events: 0, Active phone call: 0ms (0.0%)
 Screen brightnesses: dark 7s 7ms (1.5%), medium 7m 30s 861ms (98.5%)
 Kernel Wake lock "SMD_DS": 2s 368ms (3 times) realtime
 Kernel Wake lock "mmc_delayed_work": 1s 210ms (1 times) realtime
 Kernel Wake lock "SMD_RPCCALL": 56ms (435 times) realtime
 Kernel Wake lock "power-supply": 575ms (4 times) realtime
 Kernel Wake lock "radio-interface": 3s 1ms (3 times) realtime
 Kernel Wake lock "ApmCommandThread": 4ms (10 times) realtime
 Kernel Wake lock "ds2784-battery": 2s 6ms (21 times) realtime
 Kernel Wake lock "msmfb_idle_lock": 14ms (2273 times) realtime
 Kernel Wake lock "kgsl": 51s 482ms (613 times) realtime
 Kernel Wake lock "rpc_read": 164ms (272 times) realtime
 Kernel Wake lock "main": 7m 39s 708ms (0 times) realtime
 Total received: 0B, Total sent: 0B
 Total full wakelock time: 149ms , Total partial waklock time: 31s 14ms
 Signal levels: none 59m 57s 63ms (99.9%) 1x
 Signal scanning time: 59m 57s 63ms
 Radio types: none 641ms (0.0%) 1x, other 59m 56s 973ms (99.9%) 1x
 Radio data uptime when unplugged: 0 ms
 Wifi on: 59m 57s 709ms (99.9%), Wifi running: 22m 35s 424ms (37.6%), Bluetooth
on: 0ms (0.0%)

 Device battery use since last full charge
 Amount discharged (lower bound): 0
 Amount discharged (upper bound): 1
 Amount discharged while screen on: 1
 Amount discharged while screen off: 0

(...and lots more...)
```

and per-process information (here, showing power used by PowerTutor itself):

```
#10058:
 Wake lock window: 5s 71ms window (1 times) realtime
 Proc edu.umich.PowerTutor:
 CPU: 11s 750ms usr + 4s 530ms krn
 1 proc starts
 Apk edu.umich.PowerTutor:
 Service edu.umich.PowerTutor.service.UMLoggerService:
 Created for: 4m 4s 750ms uptime
 Starts: 1, launches: 1
```

In principle, one might create tools that use this output – or perhaps steal a peek at the data used by the Settings application – to create something a bit more developer-friendly.

Keyword Index