

78. A Guide to the Android Studio Profiler

Introduced in Android Studio 3.0, the Android Profiler provides a way to monitor CPU, networking and memory metrics of an app in realtime as it is running on a device or emulator. This serves as an invaluable tool for performing tasks such as identifying performance bottlenecks in an app, checking that the app makes appropriate use of memory resources and ensuring that the app does not use excessive networking data bandwidth. This chapter will provide a guided tour of the Android Profiler so that you can begin to use it to monitor the behavior and performance of your own apps.

78.1 Accessing the Android Profiler

The Android Profiler appears in a tool window which may be launched either using the *View -> Tool Windows -> Android Profiler* menu option or via any of the usual toolbar options available for displaying Android Studio Tool windows. Once displayed, the Profiler Tool window will appear as illustrated in [Figure 78-1](#):

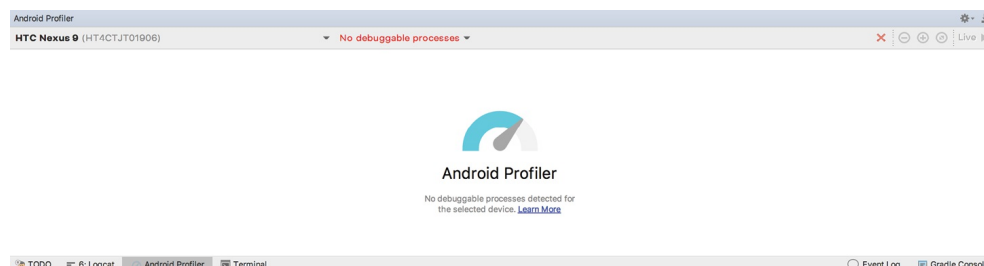


Figure 78-1

In the above figure, no processes have been detected on any connected devices or currently running emulators. To see profiling information, an app will need to be launched. Before doing that, however, it may be necessary to configure the project to enable advanced profiling information to be collected.

78.2 Enabling Advanced Profiling

If the app is built using an SDK older than API 26, it will be necessary to build the app with some additional monitoring code inserted during compilation in order to be able to monitor all of the metrics supported by the Android

Profiler. To enable advanced profiling, begin by editing the build configuration settings for the build target using the menu in the Android Studio toolbar shown in [Figure 78-2](#):

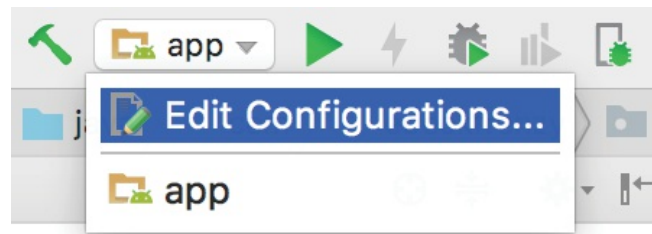


Figure 78-2

Within the Run/Debug configuration dialog, select the *Profiling* tab and enable the *Enable advanced profiling* option before clicking on the *Apply* and *OK* buttons.

78.3 The Android Profiler Tool Window

Once an app is running it can be selected from the device and app selection menus (marked A and B in [Figure 78-3](#)) within the Android Profiling tool window to begin monitoring activity.

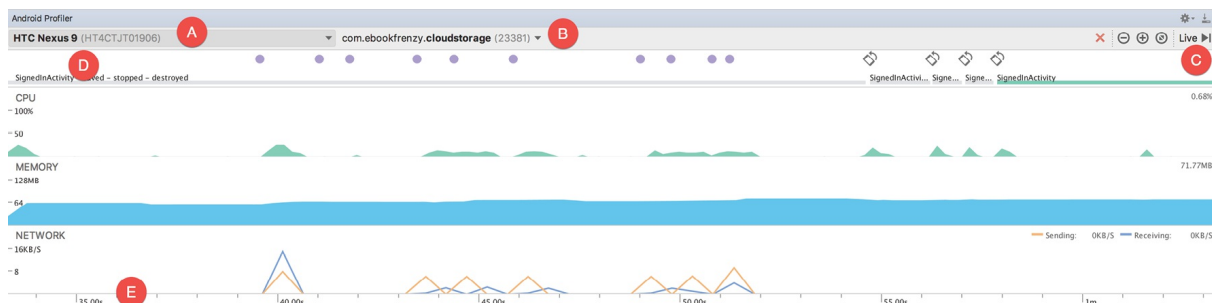


Figure 78-3

The window will continue to scroll with the latest metrics unless it is paused using the *Live* button (C). Clicking on the button a second time will jump to the current time and resume scrolling. Horizontal scrolling is available for manually moving back and forth within the recorded time-line.

The top row of the window (D) is the *event time-line* and displays changes to the status of the app's activities together with other events such as the user touching the screen, typing text or changing the device orientation. The bottom time-line (E) charts the elapsed time since the app was launched.

The remaining timelines show realtime data for CPU, memory and network usage. Hovering the mouse pointer over any point in the time-line (without

clicking) will display additional information similar to that shown in [Figure 78-4](#).

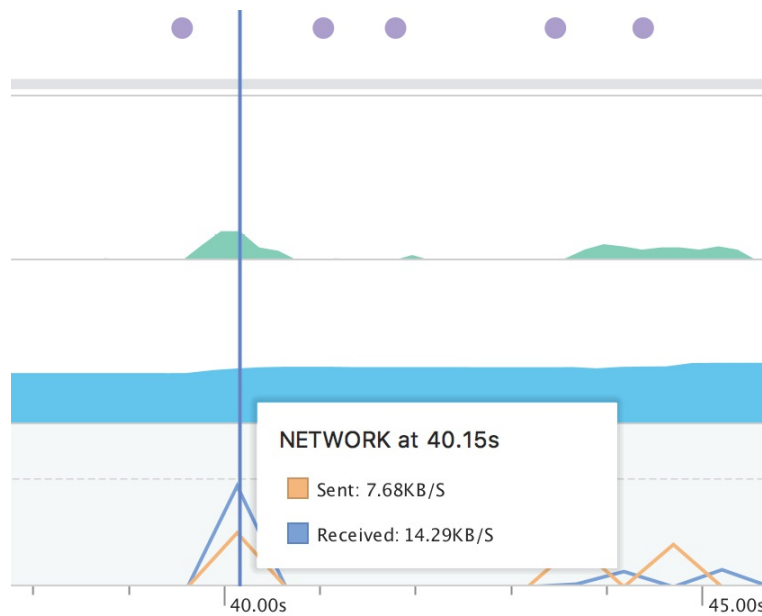


Figure 78-4

Clicking within the CPU, memory or networking timelines will display the corresponding profiler window, each of which will be explored in the remainder of this chapter.

78.4 The CPU Profiler

When displayed, the CPU Profiler window will appear as shown in [Figure 78-5](#). As with the main window, the data is displayed in realtime including the event time-line (A) and a scrolling graph showing CPU usage (B) in realtime for both the current app and a combined total for all other processes on the device:

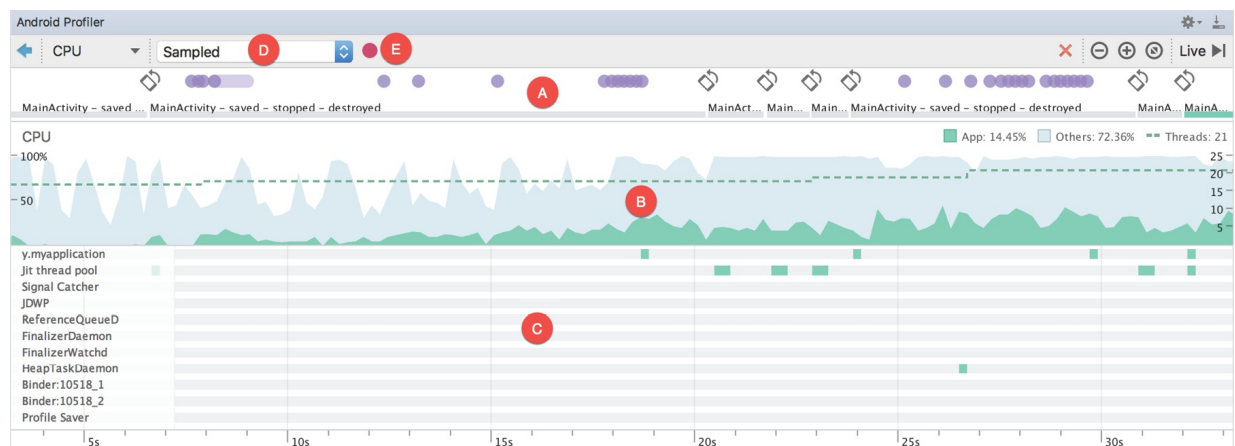


Figure 78-5

Located beneath the graph is a list of all of the threads associated with the current app (C). Referred to as the *thread activity timeline*, this also takes the form of a scrolling time-line displaying the status of each thread as represented by colored blocks (green for active, yellow for active but waiting for a disk or network I/O operation to complete or gray if the thread is currently sleeping).

The CPU Profiler supports two types of method tracing (in other words profiling individual methods within the running app). The current tracing type, either sampled or instrumented, is selected using the menu marked D. The two tracing types can be summarized as follows:

- **Sampled** – Captures the method call stack at frequent intervals to collect tracing data. While less invasive than instrumented tracing, sampled tracing can miss method calls if they occur during the intervals between captures. Snapshot frequency may be changed by selecting the *Edit configurations...* button within the type selection menu and creating new custom trace types.
- **Instrumented** – Traces the beginning and ending of all method calls performed within the running app. This has the advantage that no method calls are missed during profiling, but may impact app performance due to the overhead of tracing all method calls, resulting in misleading performance data.

Method tracing does not begin until the record button (E) is clicked and continues until the recording is stopped. Once recording completes, the Profiler tool window will display the method trace in *top down* format as shown in [Figure 78-6](#) including information execution timings for the methods.

The trace results may be viewed in Top Down, Bottom Up, Call Chart and Flame Chart modes, each of which can be summarized as follows:

- **Top Down** – Displays the methods called during the trace period in a hierarchical format. Selecting a method will unfold the next level of the hierarchy and display any methods called by that method:

Top Down Bottom Up Call Chart Flame Chart				Wall Clock Time			
Name	Self (us)	%	Children (us)	%	Total (us)	%	
JDWP	3,498,238	99.92%	2,888	0.08%	3,501,126	100.00%	
dispatch() (org.apache.harmony.dalvik.ddmc.DdmServer)	1,937	0.06%	951	0.03%	2,888	0.08%	
get() (java.util.HashMap)	32	0.00%	418	0.01%	450	0.01%	
getEntry() (java.util.HashMap)	120	0.00%	293	0.01%	413	0.01%	
equals() (java.lang.Integer)	173	0.00%	5	0.00%	178	0.01%	
intValue() (java.lang.Integer)	5	0.00%	0	0.00%	5	0.00%	
singleWordWangJenkinsHash() (sun.misc.Hashing)	78	0.00%	31	0.00%	109	0.00%	
hashCode() (java.lang.Integer)	31	0.00%	0	0.00%	31	0.00%	
indexOf() (java.util.HashMap)	6	0.00%	0	0.00%	6	0.00%	
getValue() (java.util.HashMap\$HashMapEntry)	5	0.00%	0	0.00%	5	0.00%	
handleChunk() (android.ddm.DdmHandleProfiling)	35	0.00%	210	0.01%	245	0.01%	
valueOf() (java.lang.Integer)	181	0.01%	58	0.00%	239	0.01%	
<init>() (org.apache.harmony.dalvik.ddmc.Chunk)	13	0.00%	4	0.00%	17	0.00%	

Figure 78-6

- **Bottom Up** – Displays an inverted hierarchical list of methods called during the trace period. Selecting a method displays the list of methods that called the selected method:

Top Down Bottom Up Call Chart Flame Chart				Wall Clock Time			
Name	Self (us)	%	Children (us)	%	Total (us)	%	
JDWP	3,498,238	99.92%	2,888	0.08%	3,501,126	100.00%	
dispatch() (org.apache.harmony.dalvik.ddmc.DdmServer)	1,937	0.06%	951	0.03%	2,888	0.08%	
get() (java.util.HashMap)	32	0.00%	418	0.01%	450	0.01%	
dispatch() (org.apache.harmony.dalvik.ddmc.DdmServer)	32	0.00%	418	0.01%	450	0.01%	
getEntry() (java.util.HashMap)	120	0.00%	293	0.01%	413	0.01%	
get() (java.util.HashMap)	120	0.00%	293	0.01%	413	0.01%	
handleChunk() (android.ddm.DdmHandleProfiling)	35	0.00%	210	0.01%	245	0.01%	
dispatch() (org.apache.harmony.dalvik.ddmc.DdmServer)	35	0.00%	210	0.01%	245	0.01%	
valueOf() (java.lang.Integer)	181	0.01%	58	0.00%	239	0.01%	
dispatch() (org.apache.harmony.dalvik.ddmc.DdmServer)	181	0.01%	58	0.00%	239	0.01%	
equals() (java.lang.Integer)	173	0.00%	5	0.00%	178	0.01%	
handleMPSS() (android.ddm.DdmHandleProfiling)	9	0.00%	119	0.00%	128	0.00%	
startMethodTracingDdms() (android.os.Debug)	7	0.00%	112	0.00%	119	0.00%	
startMethodTracingDdms() (dalvik.system.VMDebug)	111	0.00%	1	0.00%	112	0.00%	
singleWordWangJenkinsHash() (sun.misc.Hashing)	78	0.00%	31	0.00%	109	0.00%	
<init>() (java.lang.Integer)	25	0.00%	33	0.00%	58	0.00%	

Figure 78-7

- **Call Chart** – Provides a graphical representation of the method trace list where the horizontal axis represents the start, end and duration of the method calls. In the vertical axis, each row represents methods called by the method above. Methods contained within the app are colored green, API methods orange and third-party methods appear in blue:

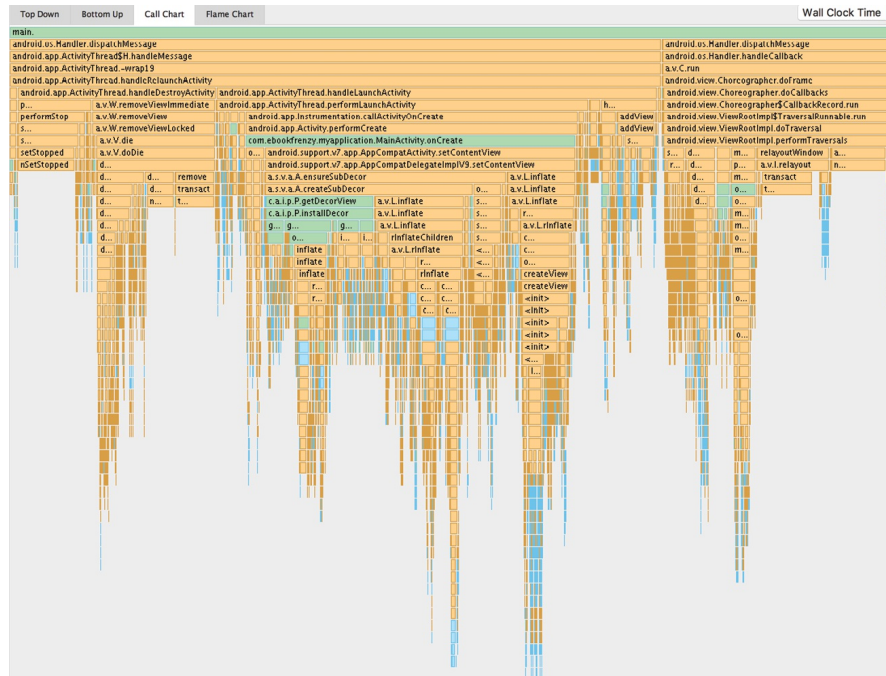


Figure 78-8

- **Flame Chart** – Provides an inverted graphical representation method trace list where each method is sized on the horizontal axis based on the amount of time the method was executing relative to other methods. Wider entries within the chart represent methods that used the most execution time relative to the other methods making it easy to identify which methods are taking the most time to complete. Note that method calls that have matching call stacks (in other words situations where the method was called repeatedly as the result of the same sequence of preceding method calls) are combined in this view to provide an overall representation of the method's performance during the trace period:

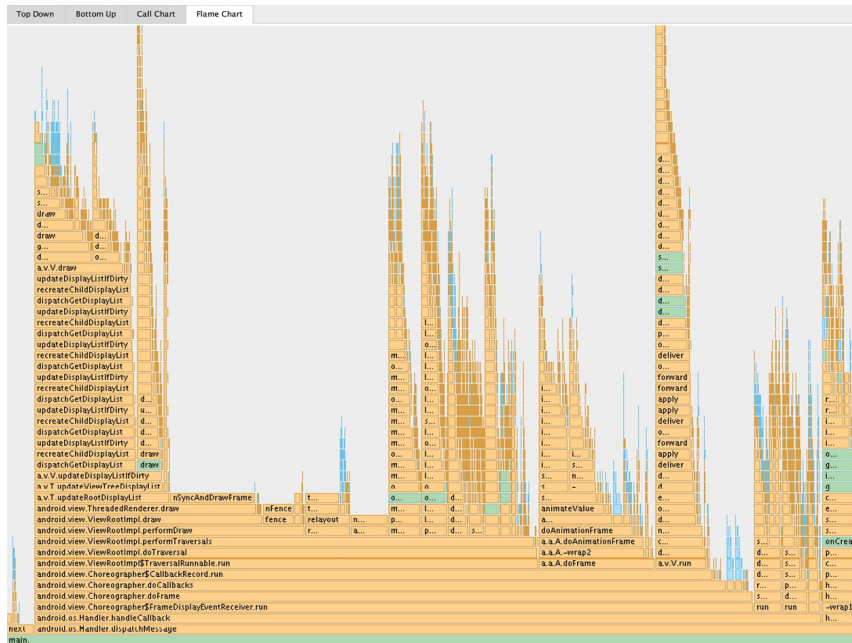


Figure 78-9

Right-clicking on a method entry in any of the above views provides the option to open the source code for the method in a code editing window.

78.5 Memory Profiler

The memory profiler is displayed when the memory time-line is clicked within the main Android Profiler Tool window and appears as shown in [Figure 78-10](#):

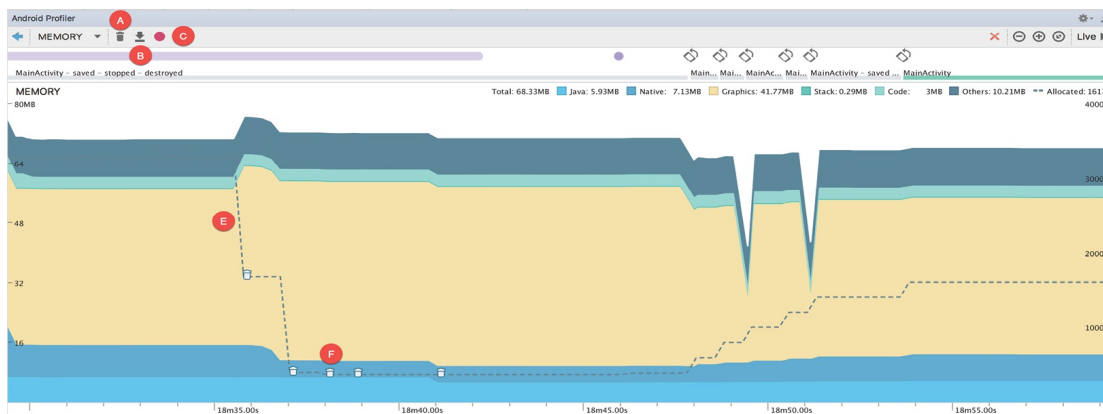


Figure 78-10

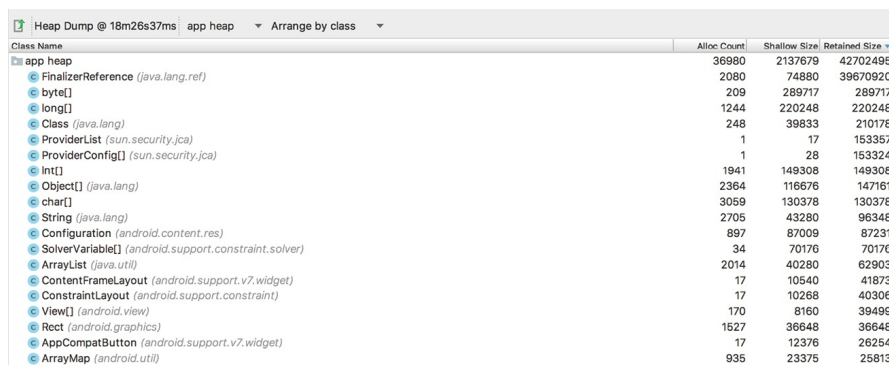
The memory time-line shows memory allocations relative to the scale on the right-hand side of the time-line for a range of different categories as indicated by the color key. The dashed line (E) represents the number of objects

allocated for the app relative to the scale on the left-hand side of the time-line graph.

The trash can icons (F) indicate *garbage collection* events. A garbage collection event occurs when the Android runtime decides that an object residing in memory is no longer needed and automatically removes it to free memory.

In addition to the usual timelines, the window includes buttons to force garbage collection events (A) and to capture a heap dump (B).

A heap dump ([Figure 78-11](#)) lists all of the objects within the app that were using memory at the time the dump was performed showing the number of instances of the object in the heap (allocation count), the size of all instances of the object (shallow size) and the total amount of memory being held by the Android runtime system for those objects (retained size).



The screenshot shows the 'Heap Dump @ 18m26s37ms' window in Android Studio. The title bar indicates 'app heap' and 'Arrange by class'. The table lists various classes with their allocation count, shallow size, and retained size. The classes are sorted by class name.

Class Name	Alloc Count	Shallow Size	Retained Size
app heap	36980	2137679	42702495
FinalizerReference (java.lang.ref)	2080	74880	39670920
byte[]	209	289717	289717
long[]	1244	220248	220248
Class (java.lang)	248	39833	210178
ProviderList (sun.security.jca)	1	17	153357
ProviderConfig[] (sun.security.jca)	1	28	153324
Int[]	1941	149308	149308
Object[] (java.lang)	2364	116676	147161
char[]	3059	130378	130378
String (java.lang)	2705	43280	96348
Configuration (android.content.res)	897	87009	87231
SolverVariable[] (android.support.constraint.solver)	34	70176	70176
ArrayList (java.util)	2014	40280	62903
ContentFrameLayout (android.support.v7.widget)	17	10540	41873
ConstraintLayout (android.support.constraint)	17	10268	40306
View[] (android.view)	170	8160	39499
Rect (android.graphics)	1527	36648	36648
AppCompatButton (android.support.v7.widget)	17	12376	26254
ArrayMap (android.util)	935	23375	25813

Figure 78-11

Double clicking on an object in the heap list will display the Instance View panel (marked A in [Figure 78-12](#)) displaying a list of instances of the object within the app. Selecting an instance from the list will display the References panel (B) listing where the object is reference. [Figure 78-12](#), for example shows that a String instance has been selected and is listed as being referenced by a variable named *myString* located in the MainActivity class of the app:

Instance	Depth	Shallow Size	Retained Size
String@314658816 (0x12c15000) "This is an Example String in My App!"	0	16	2134
String@314687984 (0x12c1c1f0) ". If the resource you are trying to use i	0	16	386
String@314587648 (0x12c03a00) "Studio Profilers encountered an une	0	16	384
String@314985456 (0x12c64bf0) "aq:native-post-ime:com.ebookfrenzy	5	16	198
String@314985664 (0x12c64cc0) "aq:native-pre-ime:com.ebookfrenzy	4	16	196
String@314799680 (0x12c37640) "aq:pending:com.ebookfrenzy.myappl	3	16	182
String@315482512 (0x12cde190) "aq:ime:com.ebookfrenzy.myappli	6	16	174
String@315483216 (0x12cde450) "com.google.android.inputmethod.lai	1	16	166
String@314693600 (0x12c1d7e0) "com.ebookfrenzy.myapplication/cor	3	16	160
String@314693440 (0x12c1d740) "/data/app/com.ebookfrenzy.myappli	6	16	146
String@315528816 (0x12ce9670) "/data/local/tmp/perfd/cache/comple	0	16	142
String@314739008 (0x12c28940) "Dalvik/2.1.0 (Linux; U; Android 7.1.1	4	16	136
String@314598784 (0x12c06580) "/data/user_de/0/com.ebookfrenzy.n	2	16	128
String@314599296 (0x12c06780) "/system/priv-app/SettingsProvider/	7	16	124
String@314599040 (0x12c06680) "/system/priv-app/SettingsProvider/	7	16	124
String@314598528 (0x12c06480) "android.security.net.config.RootTru	7	16	124
String@314845912 (0x12c42ad8) "res/drawable/action_bar_item_back	9	16	120
String@314843512 (0x12c42178) "/data/app/com.ebookfrenzy.myappli	3	16	118
String@314843872 (0x12c422e0) "/data/app/com.ebookfrenzy.myappli	6	16	118
String@314873744 (0x12c49790) "/data/app/com.ebookfrenzy.myappli	6	16	118
String@314843752 (0x12c42268) "/data/app/com.ebookfrenzy.myappli	6	16	116

Reference	Depth	Shallow Size	Retained Si...
String@314658816 (0x12c15000)	0	16	2134
myString in MainActivity@314905856 (0x12c51500)	3	256	4031

Figure 78-12

Right-clicking on the reference would provide the option to go to the MainActivity class in the heap list, or jump to the source code for that class.

The *Record memory allocations* button (marked C in [Figure 78-10](#) above) will record memory allocations until the button is clicked a second time to stop recording. Once recording is stopped, a list of memory allocations will appear showing allocation count and shallow size values for each class as shown in [Figure 78-13](#):

Class Name	Alloc Count	Shallow Size
default heap	4471	168000
Object[] (java.lang)	921	23384
byte[]	2	20504
ArrayList (java.util)	738	17712
FinalizerReference (java.lang.ref)	431	17240
RenderNodeAnimator (android.view)	180	15840
float[]	126	12624
VirtualRefBasePtr (com.android.internal.util)	420	6720
String (java.lang)	83	6296
Paint (android.graphics)	61	5368
CanvasProperty (android.graphics)	240	3840

Figure 78-13

Selecting a class from the list will display the Instance View panel listing instances of that class. When an instance is selected, the Call Stack panel will populate with the method trace information for the instance. In [Figure 78-14](#), for example, the Call Stack panel indicates that a String object instance was allocated in a method named *myMethod* located in the MainActivity class which was, in turn, triggered by an *onClick* event in the main thread:

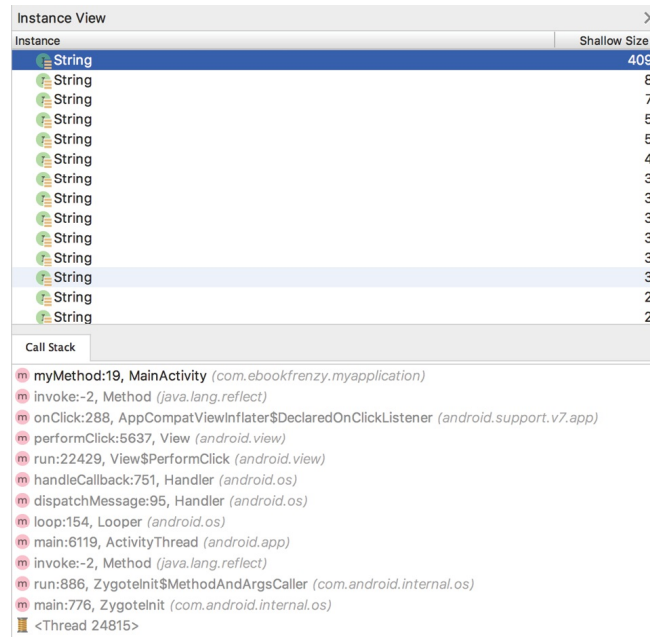


Figure 78-14

78.6 Network Profiler

The Network Profiler is the least complex of the tools provided by the Android Profiler. When selected the Network tool window appears as shown in [Figure 78-15](#):

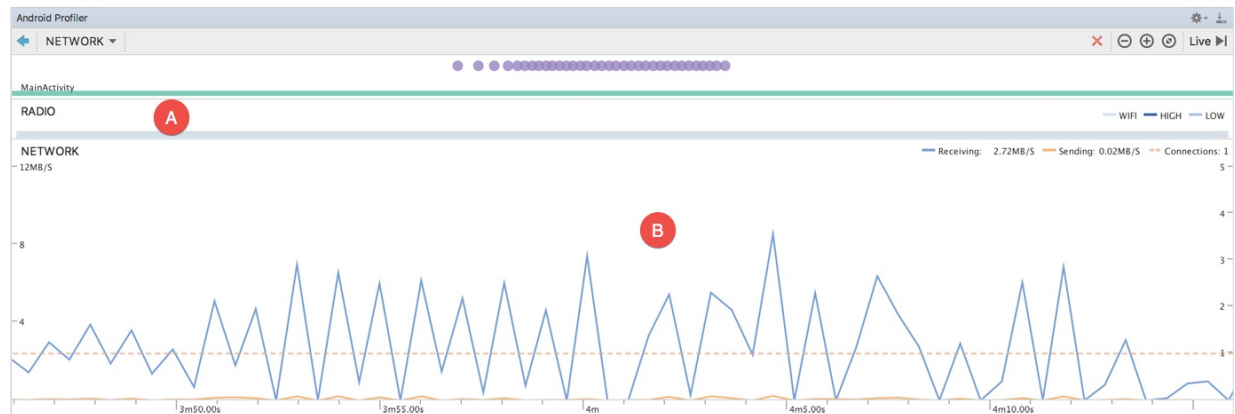


Figure 78-15

In common with the other profiler windows, the Network Profiler window includes an event time-line. The Radio time-line (marked A in [Figure 78-15](#)) shows the power status of the radio relative to the Wi-Fi connection if one is available.

The time-line graph (B) includes sent and received data and a count of the number of current connections. At time of writing, the Network Profiler is only able to monitor network activity performed as a result of HttpURLConnection and OkHttp based connections.

To view information about the files sent or received, click and drag on the time-line to select a period of time. On completing the selection, the panel labeled A in [Figure 78-16](#) will appear listing the files. Selecting a file from the list will display the detail panel (B) from which additional information is available including response, header and call stack information:

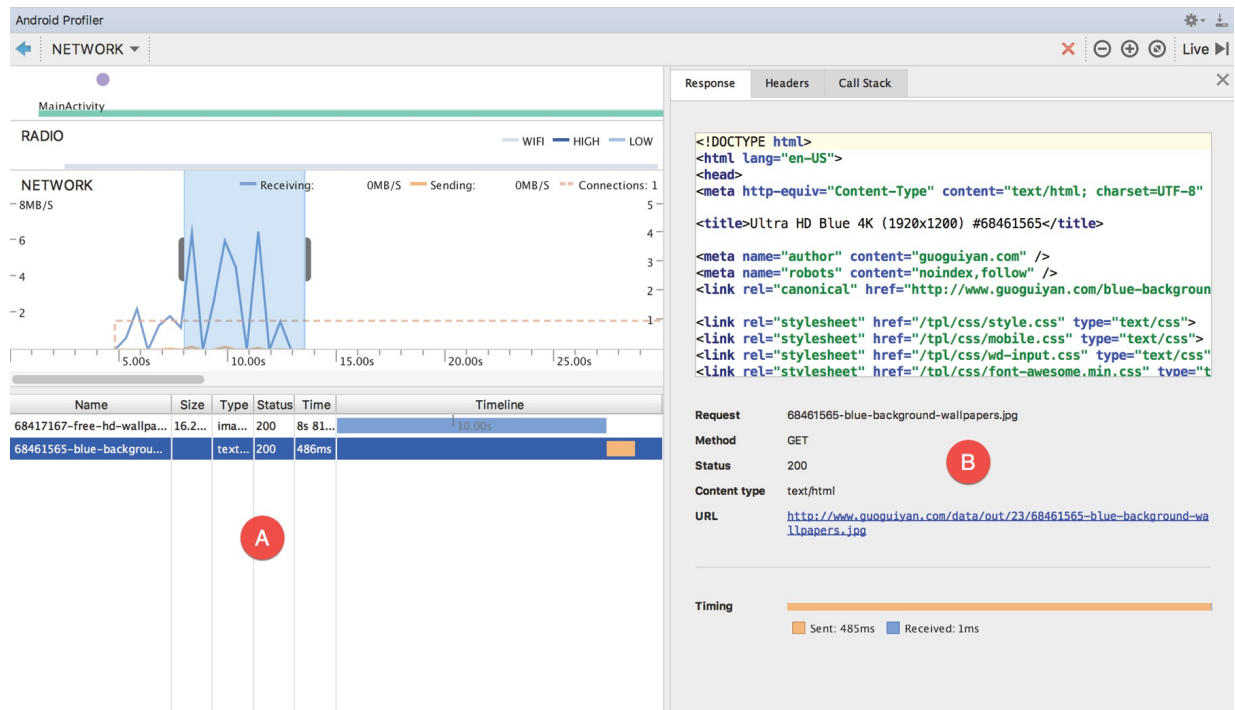


Figure 78-16

78.7 Summary

The Android Profiler monitors the CPU, memory and network resource usage of apps in realtime providing a visual environment in which to locate memory leaks, performance problems and the excessive or inefficient transmission of data over network connections. Consisting of four different profiler views, the Android Profile allows detailed metrics to be monitored, recorded and analyzed.