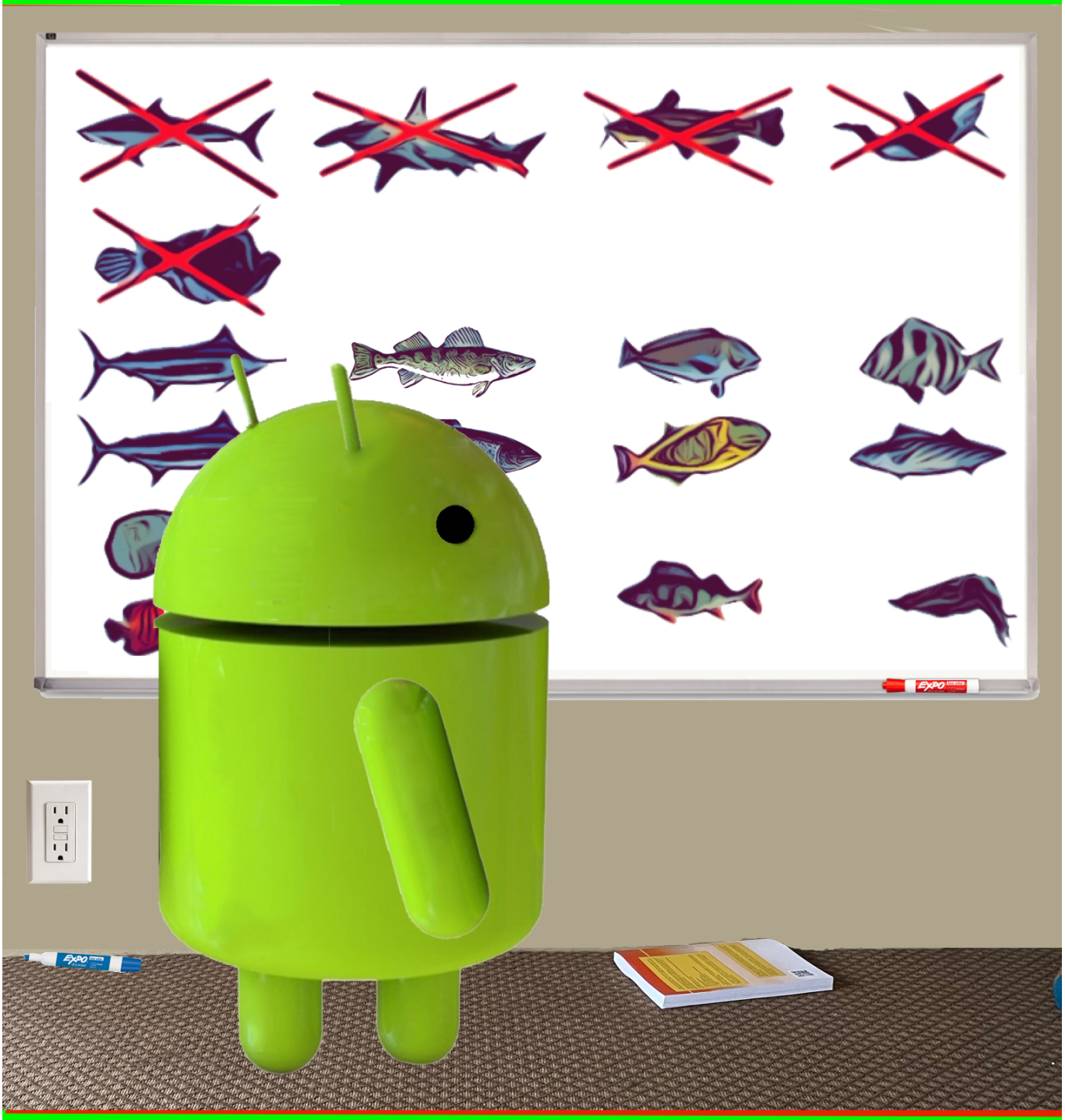
Android Internals:

Volume I - The Power User's View



By Jonathan Levin

Android Internals

A Confectioner's Cookbook

Volume I: The Power User's View

Version 2.0 - Revised and expanded with updates for Android 11

By Jonathan Levin

North Castle, NY

In memoriam: Frank R. Dye. *I missed you by a day, and will miss you for a lifetime*

Table of contents

Chapter 0: About This Book

Chapter 1: A first glance at Android's architecture

Chapter 2: Android Hardware

Chapter 3: Android Partitions & Filesystems

Chapter 4: Android Files & Directories

Chapter 5: Storage Management

Chapter 6: Android System Images & Updates

Chapter 7: The Android Boot Process

Chapter 8: User mode startup - `init` & `Zygote`

7. Android (AOSP) Daemons, at a glance

Chapter 9: The Framework Service Architecture

1. The Service Calling Pattern
    1.1. Nomenclature
    1.2. Advantages and disadvantages
    1.3. Serialization and the Android Interface Definition Language (AIDL)
2. The Binder
    2.1. A little history
    2.2. So, what, exactly, is Binder?
    2.3. Using Binder
    2.4. 8.0+: The `vndbinder` and `hwbinder`
    2.5. Tracing Binder
3. The `servicemanager`
    3.1. The `android.os.IServiceManager` Interface (Android 11)
4. `system_server`
    4.1. Handling Services
    4.2. Startup and Flow
5. A bird's eye view of framework Services
    5.1. LocalServices

Chapter 10: Device Configuration & Management

1. User Management
    1.1. The `user` service
2. Account Management
    2.1. The `accounts` database
    2.2. The `account` service
3. Configuration Settings
    3.1. `config.xml` and other files
    3.2. Overlays
    3.3. The `settings` service
    3.4. The `device_config` service
    3.5. Server Configurable Flags
    3.6. The `.../etc/sysconfig` directories
    3.7. The `system_config` service (Android 11)

Chapter 11 : Android Through a Linux Lens

1. Processes and threads in Android (and Linux)
2. `/proc`, revisited
    2.1. The symlinks: cwd, exe, root
    2.2. The `cmdline` and `comm`
    2.3. fd
    2.4. fdinfo
    2.5. status
3. Process Management

Chapter 12: Logging, Statistics & Monitoring

Chapter 13: Power Management

About This Book

1. Overview

This is the new, and greatly revised edition of "Android Internals". The original work, published in 2015, covered up to Android L. Since then, I constantly kept it updated with incremental modifications as Android progressed. Over time, these changes amassed, and required revisions as some features were no longer supported. When my book gained world fame but sales crashed (thanks to the CIA and the reckless WikiLeaks), I knew a revision would be a matter of time.

If you got this book, no doubt you recognize the importance of Android. From a start-up started back in 2003, it has been assimilated by Google, and morphed into one of its largest arms. Taking on Apple's iOS head on (some would say, too closely), it has not only achieved hegemony over mobile operating systems worldwide (with a staggering 82% of the market persistently maintained) but has also permeated other platforms, becoming an operating system for wearable devices, TVs, and embedded devices.

Android is open source and freely available, meaning anyone can get it, and adopt it to any platform - indeed, it owes its overwhelming popularity to this. It was surprising, however, that over seven years after inception, no book to date has taken on the task of documenting and explicating its internals. A previous work on the subject - Embedded Android: Porting, Extending, and Customizing, by Karim Yaghmour - provides a good deal of detail about the general structure of the OS, but focuses on building and adapting the sources to new platforms, and stops shy of describing the structure of the operating system itself. In fact, in his "Internals Primer", Yaghmour states that "Fully understanding the internals of Android's system services is like trying to swallow a whale".

The analogy is very much an understatement. Which is why this work requires not one, but multiple volumes. The first (the one you are reading), focuses on Android from the perspective of the power user or administrator. In it, I try to tackle various aspects of the operating system - its design, filesystem structure, boot sequence, and native services, along with the Linux foundations and legacies which affect the operation. All this, without going into code, and trying to provide an illustrated, conceptual view as possible. This book can be considered, in a sense, a successor to Yaghmour's work, which remains a great resource and a recommended read.

The second volume of this work (which finally sees print, a fashionable five years later) dives far deeper, and looks at the structure of Android's frameworks - which is where its appeal to developers lies: Through a rich set of Java-level frameworks, developers obtain powerful abstractions of input devices, sensors, graphics and what not. All these abstractions, come at the price - the complexity that lies "under the hood" - which most developers are quite blissfully ignorant of (and would likely prefer to stay this way). There is no knowledge that is not power, however, and so deep familiarity with the frameworks is instrumental for anyone dealing with the low level implementations, and customizations for performance, hardware or security.

And, since you're reading this "second edition", you probably know by now that the series has been expanded to three volumes, after all. Volume III, which I had originally mulled for kernel*, is now set to cover Android's security. That means that the security coverage in the previous edition of this work has been moved out of what used to be Chapter 8, into its own book.

---

* - Originally, I was foolish enough to think a third volume, dealing with Android kernel changes, would be a good idea. Android kernels, however, are 99% identical to Linux (with some `CONFIG` settings, platform drivers, and minor "Androidisms"), and that would have meant writing a full Linux Kernel book. A feat which hasn't been attempted in the past 15 years. But never say never

Android is a constantly shifting landscape. This work was started halfway through KitKat, and was postponed several times as Android mutated further to become Lollipop (L) when the book came out. This went on and on, and after continuously posting differential updates all the way to Oreo I decided it's time for a revision, thus ending up with this "v2.0", which has been updated - and in some cases rewritten - for 11 (Q). So this book is updated till the latest and greatest.. at least at the time of publication.

The first edition of this work tried to focus more on illustrations and less on source code snippets, but this work relaxes this somewhat. Especially in cases where the source code is properly documented already. My own personal belief was and is of "Read the Source, Luke", in that source code - unlike natural language - contains (almost) no ambiguities, and is thus the right way of depicting facts. It is especially because Android's sources are available - though most people haven't gone into them as they are so overwhelming - that I allowed myself to show more source code, and leave the paths and hyperlinks to the Android source base as well.

The book continues the "hands-on" approach, taking some of the hands-on exercises from our Android training and recasting them in the form of Experiments. These are invaluable if you want to get a good sense of the topics in the relevant section. Android is a UN*X derivative (by virtue of Linux), and the only way one learns UN*X is through the fingers, and neither eyes nor ears. The experiments demonstrate many useful commands from the Android command-line-interface (CLI), and also techniques for looking deeper into the operating system. Furthermore, the experiments will likely produce different outputs on different strains of Android - which makes them worthwhile to try on your own device(s), so as to get different perspectives or implementations which may vary by vendor or OS version.

2. Quid Novi?

So much has changed and has been added over the past five years that this, for all intents and purposes, can be considered a "second edition" of Android Internals. Volume I delves deeper into hardware and vendor particularities, as well as includes chapters I had originally thought would be better off in Volume II, but now realize differently. Many topics' coverage was expanded, to the point of putting them into their own chapters (Updates, Storage Management, Logging). What was Chapter 2 (Partitions and Filesystems) is now two chapters - one for Partitioning, and one for a filesystem tour, in which every single AOSP file in `/data` is accounted for.

Whereas Volume I previously provided (in Chapter 5) a cursory glance into the many daemons and almost entirely avoided framework services (thinking I'd cover the latter in Volume II), a major change is achieving **full coverage** of every single daemon and framework service in AOSP up to Android 11. This means some 50 daemons and four times that many services! A side effect is that Chapter 5 has been removed, and now the daemonic discussions are provided (with more detail than before) in their respective subsystem chapters.

I've introduced the notion of "business cards" for daemons and services. I cover each in great detail, of course, but sometimes I believe the reader will just need access to the salient high-level details (implementing binaries or classes, files, etc). Thus, floating to the right of any detailed discussion will appear those details, in an easy to read form. Details differ between the different daemons and services. For services, I note the interface, manager and implementation classes, and where they're started from. Daemons are all started by `/init`, so that's irrelevant - but they are in separate projects, so those are indicated. For both, I note any files, directories, or socket used, and - importantly - any permissions. Where possible I also note the clients and servers, but since these can be many, I do not aim to be complete.

Another very prominent omission from this work is the removal of Chapter 8 - Security. As I mentioned - security is now handled in a third volume. It was a very hard decision to make (considering my poor track record with Volume II), but the more I thought about it the more it seemed like the only choice I could make. Android's security is unfortunately complex (arguably, more than it need be), due to its layering on numerous disparate underlying Linux facitilies. I remind my security focused readers that the first edition's coverage of security (somewhat dated, but not that bad, I hope) is still freely available.

3. Contents, at a glance

The book is designed to be read either cover-to-cover or as random, quick access. Each chapter is largely self contained, and hyperlinks on topics allow quick associative navigation when reading the book in e-Form. For print edition, relevant chapter numbers (for internal links) or URLs (for external links) are provided.

Chapter One provides an **introduction** to the operating system: Examining the evolution of the OS over its versions (since Froyo, 2.2, which was the oldest version I covered in the previous work, and up to 11). It also explains the architecture (at a high level view), and the Linux underpinnings, by traversing each layer of the Android stack. It then looks at Android derivatives, both Google's and other vendors (e.g. Amazon's FireOS).

Chapter Two is a new discussion of Android **hardware**. Although Android - like its Linux core - can run on virtually any architecture, we limit the discussion to ARMv8 - the predominant architecture running all mobile devices - which is quite diverse by itself. The chapter covers the ARM processor variants and versions, and provides an overview of Systems on Chip (SoCs), before focusing on specific vendor implementations - Qualcomm, Samsung, Huawei and MediaTek. It also discusses the important Device Tree structure, and firmware loading for SoC components. Note, that this is only a high level introduction to hardware and the Linux perspective of it. Much more detail on Android's take on hardware - namely, HAL interfaces and implementations - is saved for Volume II.

Chapter Three is the first of three dealing with storage - starting with **flash partitioning and filesystems**. We start with a recap of the GPT standard, and move to Android specific schemes - A/B slotting and Android 10's dynamic partitions. Next, the standard filesystems of AOSP - both mountable and nonmountable - are then discussed. Partitions specific to vendors are listed next, and finally the pseudo-filesystems of Linux, which Android makes heavy use of. This can be thought of an introduction to the storage subsytem of Android.

Chapter Four is a tour of **filesystem contents**, which should prove useful if you ever need to figure out what a specific system directory or file contains. This chapter is virtually all tables, with the primary aim to serve as a high-level reference to files encountered in `/system`, `/data` and a little of vendor, and pointers to where else in these books the daemons or services which use them are detailed. A few of the built-in apps data directories are also covered, which is handy if you're doing forensics.

Chapter Five concludes the discussion of storage by focusing on the **storage subsystem**. First, the specific types of mounting used - loop, bind, FUSE - and mount storage. Android's specific "types" of external storage - portable, emulated and adoptable. Next, Android's daemons are detailed - the native `vold`, and the framework services of `mount` (the `StorageManager`), `storagestats` and others. Finally, Android 10's `apex` - The Android Pony EXpress subsystem, is detailed.

Chapter Six covers Android **system images & updates**. Starting with a discussion of the Android factory and OTA images (what some refer to, albeit incorrectly, as ROMs), and how to flash them onto the device's boot partitions. It then moves on to explain the two update modes - via recovery and (for newer, slotted devices) the `update_engine`. It then wraps up with a discussion of Android's Generic System Images (GSI) and Dynamic System Updates.

Chapter Seven deals exclusively with the **boot process**. While vendor-specific, the chapter generalizes the process just enough to present a still detailed view, and then explores implementation details, such as Qualcomm's UEFI LinuxLoader, and the traditional ABoot. All flows merge at the kernel and ramdisk loading, and the chapter also examines taking apart and rebuilding said ramdisk, which is a crucial step in "rooting" the device. Finally, the AOSP/bootloader two way communication is explored - via the Bootloader Control Block (BCB) in the `misc` partition, and the numerous `androidboot` kernel command-line arguments.

Chapter Eight is dedicated almost entirely to **user mode startup** - primarily, `/init`. This, like its UN*X namesake, is responsible for starting up the system in user mode. As such, it is the direct continuation of Chapter Seven, which ends with the kernel/ramdisk. The process of startup is explained in detail, through examination of the `/init.rc` file syntax. Other roles of `/init`, such as maintaining system properties and watching for hardware changes (as `ueventd`) are detailed as well. `/init` spawns numerous AOSP and vendor daemons, and so the chapter lists those of AOSP, indicating where each is described in the detail it deserves. One such daemon - `zygote` - is reviewed therein from the Linux perspective, and will be revisited from the developer's perspective in Volume II.

Chapter Nine provides a gentle introduction to Android's **framework service architecture**, by explaining the roles of the `servicemanager` and `system_server` processes, which together form the fulcrum on top of which all of Android's frameworks rest. Binder, the elephant in the chapter, is described but briefly, leaving most of the meticulous detail for Volume II, but hopefully explaining just enough to provide more insight as to how Android Inter Process Communication and Remote Procedure Calls work. It continues its predecessor by looking at `zygote`'s first-born - `system_server`, examining its high-level flow. Since `system_server` is effectively the `svchost.exe` of Android, its myriad services are listed here, again with an indication of which chapter details them in depth.

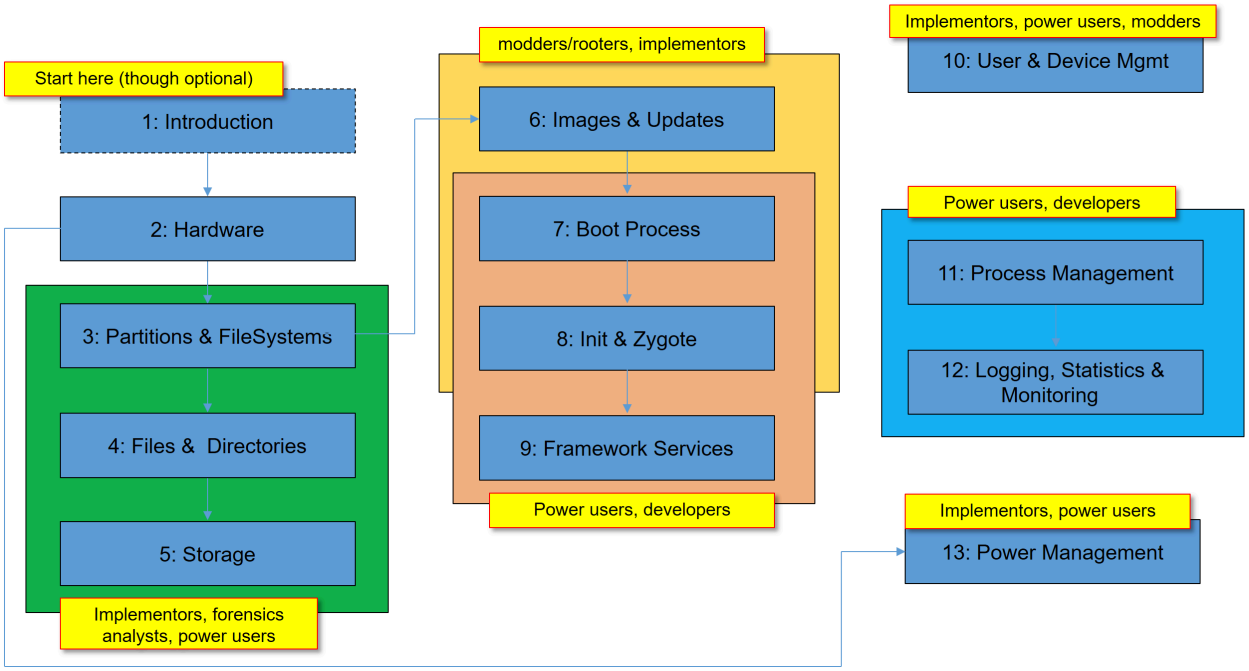
Chapter Ten focuses on **configuration and management**: Of the user profile environment, and of the device's numerous settings, whethere locally or through an admin app and Mobile Device Management (MDM). Virtually every aspect of Android - and in particular the UI - is customizable, contributing to the many "skins" and "themes" developed by vendors (and enthusiast modders) in an attempt to set their devices apart. The Settings app and a few shell commands can control some aspects of this customization, but real power and complete control is achieved by rooting, or customizing the firmware image.

Chapter Eleven is a view of Android through a Linux lens - that is, looking at Android system processes and apps through the `/proc` filesystem and Linux-level tools. This chapter is a "two-fer" in the sense that you can apply most (if not all) of the techniques shown there on your Linux system for native-level debugging. It expands on the original edition by detailing Android's process management (cgroups and task profiles), expands on the changes in the Low Memory Killer Daemon (`lmkd`), adds plentiful detail on Android's process and memory information APIs/services, and concludes with a discussion of I/O, and the new `iorapd` daemon.

Chapter Twelve - deals with **logging, statistics & monitoring**. For logging, `logd` (the server providing `logcat`) and the `DropBox` service are described in detail. Statistics is almost exclusively the domain of `statsd` and `incidentd`. Monitoring is spread out between `atrace`, Perfetto, and a case study of Qualcomm's `diag`nostics interface. The chapter concludes by formally presenting `jtrace` and `eBPF`, though usage examples abound elsewhere in the book.

Chapter Thirteen - deals with **power Management**. This was originally planned for Volume II, but makes a lot of sense here. The chapter avoids the obvious "tips to extend battery life", instead taking on discussion of the full stack. From native APIs, through the `PowerManagerService` and related services, battery and charge monitoring, processor governors, thermal management, and power statistics.

Certain chapters will appear more to different types of readers, and some chapters correspond to subsystems, so the following figure can be taken as a "reading guide" suggestion for navigating through this book, by either sequential or random access:



9

The Framework Service Architecture

The previous chapter painted only a partial picture of the runtime services in Android. The services detailed therein were all native-level processes - implemented in C/C++, and with no direct programmatic interface from the Java layer. As such, they can be classified as services which support the operating system itself. Applications, however, make use of an entirely different set of services, provided by the Dalvik-level frameworks, with special interfaces. These services have a Java language interface, and most of which run in the context of one process: `system_server`, and are reachable with the help of `servicemanager`.

We begin by examining the service managers, which provides the role of an endpoint mappers (that is, allowing service location and invocation). The services make themselves visible to clients by registering with `servicemanager` applicable to their namespace, and from that point on clients may approach that `servicemanager` and request a connection (or a handle) to the service. All framework services are invoked in the same way, and this service calling pattern, is discussed next. In particular, two key components are introduced - The Android Interface Definition Language, or **AIDL**, providing the interface (or set of APIs) exported by the services, and the `service` utility, which allows the testing and debugging of those interfaces from the command line.

The underlying transport for service (and, indeed, all inter-app) communication is Android is the **Binder** mechanism, which is accessible to applications via a character device - `/dev/binder`, `/dev/vndbinder` (for vendors) and `/dev/vndbinder` (for HIDL servers). What look like simple device nodes are, in fact, entrypoints to an elaborately designed IPC framework, which is charged with not only dispatching messages, but also with passing around objects, descriptors, and more, as well as providing reliability and security. Binder is discussed in great detailed throughout its own chapter in Volume II, but we nonetheless lay out the high level view and some salient points here.

Lastly, we take a look at `system_server` itself, which functions as the service host process, wherein most services* are implemented as threads. We detail the startup, operation, and internals of this important process. We conclude with a brief overview of the services themselves, which are cover in their respective domains' chapters throughout this work.

---

* - A few notable exceptions are SurfaceFlinger and the media services. Note that vendor services usually (and more commonly, from Android 8) run in their own process.

1. The Service Calling Pattern

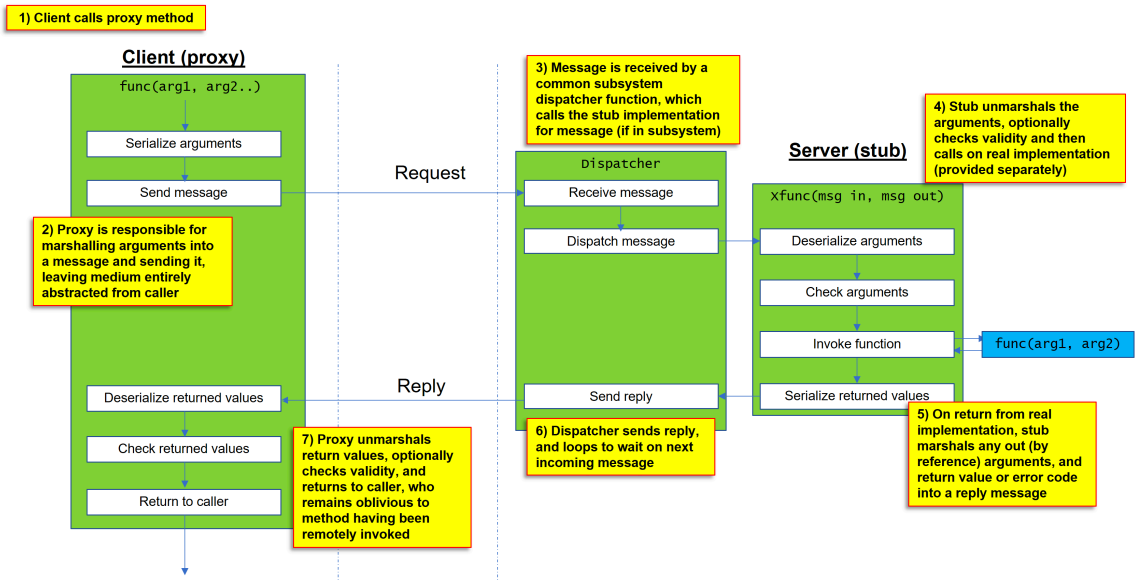
Android's framework services are (with some exceptions) implemented in `system_server` threads. Applications thus need to rely on Inter-Process Communication (IPC) in order to invoke them. This is where the **Binder**, Android's proprietary IPC mechanism, comes into play. Applications need to call on the Binder in their own process to obtain an endpoint descriptor, which is then connected to the remote service. Methods can then be invoked through IPC messages, through a pattern known as **Remote Procedure Call** (RPC).

1.1. Nomenclature

The terms IPC and RPC are often used interchageably, though often incorrectly. Because both terms are fundamental in the context of Android services, it's worth clarifying the difference:

- **Inter Process Communication (IPC)** is a blanket term for all forms of communication between processes. These include various forms of message passing, but also shared resources (most notably, shared memory), along with synchronization objects (mutexes and the like), meant to ensure safety in concurrent access to shared resources (i.e. prevent data corruption which occurs when two writers attempt to modify the same data item, or race conditions between readers and writers).
- **Remote Procedure Call (RPC)** is a specific term for a method of IPC, which hides the actual communication inside procedure (method) calls. The client calls a local method, which in turn is responsible for transparently handling the IPC with the remote server - which may at times be on a different machine. The method serializes its arguments into a message, which is then transported to the server's method, where the arguments are deserialized, acted upon, and the same occurs (in reverse) for passing the return values of the method, if any.

Figure 9/1-1: The generalized RPC architecture



Thus, any RPC mechanism is also an IPC mechanism (the former being a special case of the latter), but not vice versa. Android's service calling pattern implements RPC, as we discuss and detail in this section. Table 9/1-2 compares the RPC mechanisms used in contemporary OSes:

Table 9/1-2: Comparison of RPC mechanisms in common operating systems

OS	Mechanism	Scope	Locator	Preprocessor	Transport
UN*X	SunRPC	Local/Remote	portmapper	rpcgen	UDP,TCP
Windows	MSRPC	Local/Remote	rpcss	MIDL	TCP,HTTP
OS X/iOS	Mach	Local (Remote)	launchd (mach_init)	mig	Mach messages
Android	Binder	Local	servicemanager	aidl	/dev/*binder

As shown in the table, all RPC mechanisms have common denominators, specifically:

- scope: denoting whether the RPCs are used in between hosts (remote), or only locally*
- Locator: The server providing the directory lookup functionality, for locating services
- Preprocessor: The tool used to generate the serialization and deserialization code for messages
- Transport: The medium for message passing

1.1.1. The Endpoint Locator

For RPC to work, the client must have a way to locate the server providing the desired service, which is why an **endpoint locator** is used. The locator is a third party, which maintains the directory of services: Servers can use it to register their offered services, and clients can use it to look them up. This is reminiscient of DNS, wherein a browser uses a logical host name rather than dealing with the IP address, but is of a local scope, and returns a handle to the service. As with DNS, the client must also have a way to locate the locator, and the simplest way to provide that is to bootstrap the process by defining the locator as "well known" (similar to DNS address hard-coding), so that all peers, clients and servers alike, are able to reach the locator a priori.

1.1.2. Transactions

Remote Procedure Calls need to be used in a such a way that both the client and server can agree on which method is requested, and with what arguments. Binder uses a simple numbering scheme, wherein the **transaction code** is some value in the range of `FIRST_CALL_TRANSACTION` (0x00000001) to `LAST_CALL_TRANSACTION` (0x00ffffff), meaning some 2^{24}-1 transaction codes are possible (with numbers above that range reserved, but used by only a few system transactions like `INTERFACE..`, `DUMP..`, `SHELL_COMMAND..`, etc). The term "transaction" emphasizes the request and reply are connected in such a way that the caller can be (generally) perceive them to be atomic (though in fact they are anything but).

The local proxy code is thus responsible for translating the method code into a transaction code, as well as serializing ("marshalling") its arguments into the Binder call. At the opposite end, the incoming message is processed so as to extract its transaction code and direct it to the right server method, along with those very same arguments, now deserialized.

The server method performs whatever processing is required, and then the roles are reversed. It is now the server code which needs to serialize any `out` arguments, and the return value (if any) into the reply message. The reply is then passed to the Binder medium, through which it makes its way back to the requestor. Once received, the serialized values are read, and the method invocation returns to the local calling code, which remains oblivious to all that transpired.

1.1.3. Interfaces

Using numeric transaction identifiers is simple and efficient, but herein lies a potential problem: A client might potentially connect to the wrong service handle, and issue a transaction with an erroneous code and incorrect arguments. This could result in a client or server crash - or, worse, unintended consequences.

Binder's solution for this is to use **Interfaces**. An interface is a reverse DNS identifier, which uniquely identifies the set of methods provided by the service (or any Binder object). This way, requests for this or another service method should consist of not only the number, but also the interface identifier. This mitigates the risk of accidental method number confusion. Interfaces can be queried through a well known `INTERFACE_TRANSACTION` code, which itself can be sent on its own (i.e. does not require the interface identifier, since it is likely not known at the time). The transaction code is `_NTF` (0x5f4e5446). The high-order byte is 0x5f ('_'), so the `INTERFACE_TRANSACTION` is (along with several other codes) in Binder's reserved transaction namespace, with no risk of any interface actually claiming this code for some other purpose.

Using the `service list` command will display all registered services, and also provide (in '[]') their interfaces. Note, that the interfaces are **not** registered with the endpoint locator: `service list` first requests `servicemanager` to list all services, and then iterates over the list, obtaining the service handle for each service, and then probing it with an `INTERFACE_TRANSACTION` code.

---

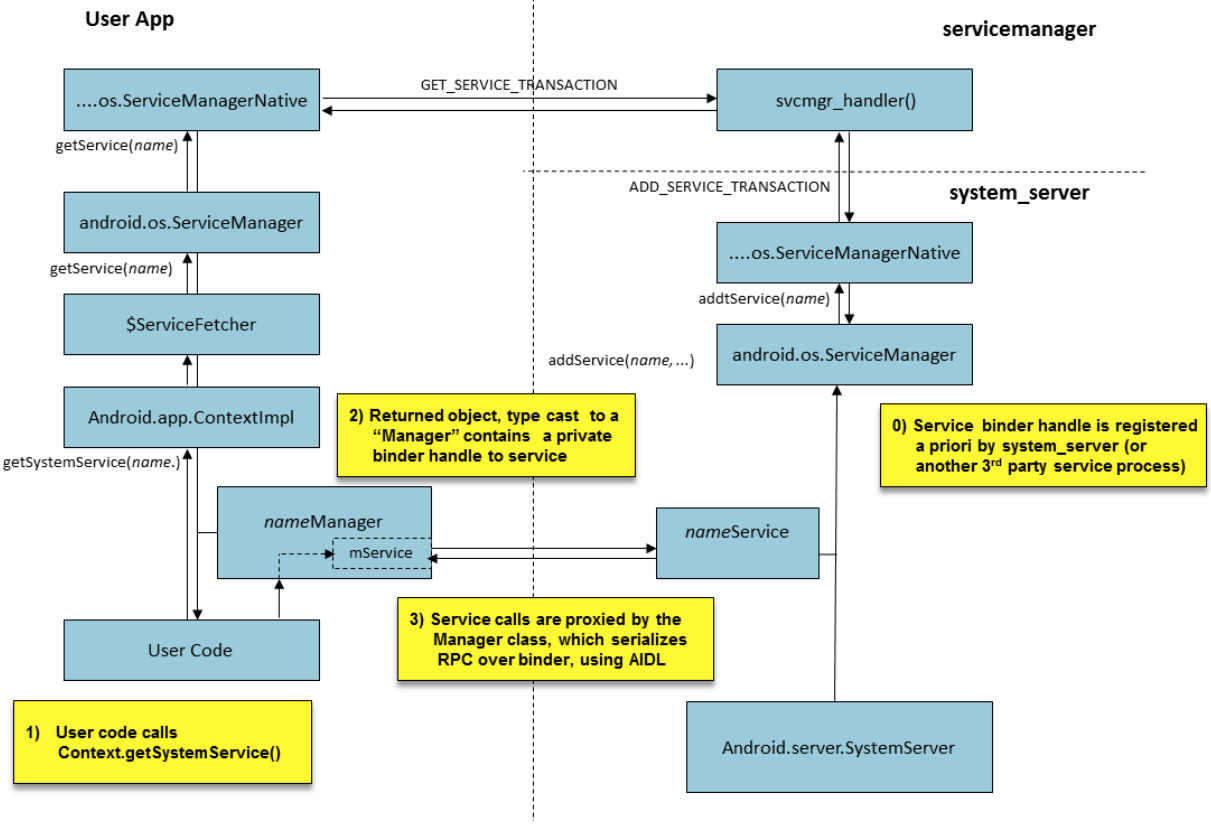
* - Android's Binder is, by design, limited to a local scope. It's possible, however, to work a local proxy to further transport the Binder RPC over a TCP or UDP socket, thus enabling remoting - highly useful capability for Remote Access or Malware.

Those services not appearing with an interface (i.e. have an empty '[]' listed in the output of `service list`) either refuse the `INTERFACE_TRANSACTION` (as they may choose to restrict their interface to `AID_SYSTEM` callers), or do not have a specific interface. In that case, they will respond only to the default transactions of `android.os.IBinder`: The `DUMP_TRANSACTION` (`_DMP`, or 0x5f444d50) of `dumpsys`, the `SHELL_COMMAND_TRANSACTION` (`_CMD`, or 0x5f434d44) of `cmd`, and a few other internal ones (all prefixed by '_').

Android developers remain blissfully oblivious to the underlying implementation of service invocation. Instead, as most Android developers are familiar with, they are required to call on the `getSystemService()` method of the `Context` object, which accepts the name of an Android system service, and returns an opaque object. The object returned can then be type cast into the specific service object, and the service methods can be invoked through it.

Figure 9/1-3 shows the general pattern followed by most service method calls. The figure is somewhat simplified (for example, the system service handles are cached), but still presents the flow. Services are registered, a priori, by the server process (commonly, `system_server`, or a 3rd party process), through a call to `android.os.ServiceManager`. Recall this class provides a Java interface to the service manager.

Figure 9/1-3: Android system service call pattern



1.2. Advantages and disadvantages

The system service architecture of Android follows a generic local client/server pattern, common to other OSes, such as iOS. Though iOS has no Binder, it uses its own implementation of a message passing architecture, called Mach messages. The role of `servicemanager` (i.e. the endpoint matter) is assumed by `iOS`'s `launchd` process, which (among other things) also handles the traditional PID 1 roles that Android's `/init` does.

A disadvantage which quickly stands out in this architecture is the overhead of IPC, particularly the need to serialize and deserialize messages, as well as the context switch required when alternating between the processes. This disadvantage does have a noticeable performance impact.

Given such a considerable disadvantage, it must be offset by advantages greater or equal in magnitude - and indeed, it is: Aside from the cleaner design and separation of privileges which follows, a client/server architecture gains security as a corollary. The client process - which is, by definition, an untrusted user app, is entirely devoid of any permissions, and therefore relies entirely on service calls to perform any operations. At the native level, this means that an app can be run sandboxed, without any access to devices and datastores, if any. Indeed, this is the case in iOS (wherein apps are "jailed"), though Android relies (for most processes) on filesystem permissions to deny access.

The server processes are trusted, and expected to perform all security checks, ensuring the client has the necessary permissions before agreeing to serve the request. Once again, the two arch rivals are similar here, with iOS relying on entitlements, (embedded in the binary's code signature), and Android on the application's Manifest file. In both cases, the permissions are declared outside of the application's runtime scope - i.e. they can be verified when installed (or, in iOS's case, when Apple vets the app), but cannot be modified by the App: Specifically, iOS's Entitlements are stored in kernel space (as part of the cached code signature blob), whereas Android's permissions are maintained by the `PackageManager`.

1.3. Serialization and the Android Interface Definition Language (AIDL)

In design pattern parlance, the object obtained from `getSystemService` serves as a **Proxy**: Internally, it holds a reference to the actual service, which it obtains over a Binder call. The methods exported by the object are, for the most part, merely stubs, which take their arguments, and serialize them into a Binder message, referred to as a `Parcel`. The methods and objects serializable in this way are specified using AIDL. AIDL isn't really a language, per se. It's essentially a derivative of Java which is understood by the `aidl` SDK utility, which is invoked in the build process when .aidl files are encountered. The `aidl` automatically generates the Java source code required to serialize any parameters into a Binder message, and extract the return value from it. The code is "boilerplate" - i.e. it can be automatically generated from the definition files and is guaranteed to compile cleanly. A sample .aidl file is shown in Listing 9/1-4:

Listing 9/1-4: A sample .aidl file

```
package com.NewAndroidBook.example;  // Creates java directory structure
import  com.NewAndroidBook.whatever; // Dependencies, if any

interface ISample {

 // Published interface - will be shown as com.NewAndroidBook.example.ISample
 // The numbers are the ones used when serializing (and using service call)

 /* 1 */ void    someMethod   (int    someArg); // no return value, integer argument
 /* 2 */ boolean anotherMethod(String someArg); // returns boolean, string argument

 // AIDL methods are commonly incrementally numbered from 1, but using '=' and the method number.
 // it is possible to assign numbers. Although this helps version compatibility, it is rarely used.

 /* 4 */  void exampleNumberedMethod(Byte[] anotherArg) = 4;

 // ... etc.. etc..

}
```

As you can see, an .aidl is somewhat similar to a header file, in that it defines methods (and possibly objects), but not their implementation. As we explore the individual framework services later in the book, you'll be able to see many more examples of actual .aidls from the AOSP.

The `aidl` tool does a marvelous job of hiding the implementation details of Android's IPC from the developers. So great a job, in fact, that most developers remain blissfully ignorant of the role of Binder, or its very existence. This work, however, recognizes the role of Binder, providing an introduction to it later in this chapter, and discussing internals in Volume II.

Power users can remain equally oblivious to Binder, especially with a powerful tool like the `service` utility, which enables the invocation of Android service methods right from the command line. A previous experiment demonstrated the basic usage of the `service` command line utility, as a method of interfacing with the `servicemanager` process. The true power of `service`, however, lies in its ability to call the services themselves, as demonstrated in the following experiment:

Experiment: Using the `service` command to call services

Calling a service is a simple enough matter - using `service call`, and specifying the service name and method number: Internally, methods are assigned numbers in order of their appearance in the service's .aidl file. Depending on the method, optional arguments may be supplied. The `service` utility supports few types (extended in Android 11). In practice, however, integers can be used for any 32 or 64-bit value, and strings - being unicode - can be used to serialize any object.

Any service retrieved by `service list` with an interface (specified in brackets) can be called on in this manner. Each interface has a corresponding .aidl file in the AOSP, wherein its methods and their arguments are clearly defined. Once you have the definitions, you can invoke any method of your choice, by figuring out its call number and passing the appropriate arguments. A few of the interesting ones are shown in Table 9/1-5:

Table 9/1-5: `service call` commands

service call...	Interface	Method	Action
`phone 2 s16 "foo" s16 "555-1234"`	`ITelephony`	`call(String callingPackage, String number)`	Place a call to specified *number*.
`statusbar 1`	`IStatusBarService`	`expandNotificationsPanel()`	Brings up notifications
`statusbar 2`		`collapsePanels()`	Hides all panels
`statusbar 9`		`expandSettingsPanel()`	Brings up settings
`dream 1`	`IDreamManager`	`dream()`	Screensaver (if configured)
`power 11`	`IPowerManager`	`isScreenOn()`	Returns 0 if screen is off, else 1

The low level call numbers assigned to methods may (and do) change between Android builds - even within the same API version (between "_r" releases). It's generally a bad idea to rely on hard coded numbers - if you intend to use these private APIs, compile alongside the updated .aidl files

Invoking calls in this way will return a result in a Parcel (the Binder term for a message). Each parcel contains, at a minimum, a 32-bit return value (0x00000000 indicating success, otherwise some error value, commonly `0xffffffff` or `0xffffffb6 ("not a data message")` if a call number is outside the defined range). Depending on the AIDL definition, what follows is either an integer value (i32), or a length specification, followed by an opaque object (usually, but not necessarily, a string). Because `service`, like Binder, has no idea of what the opqaue object is, it will display the result in a manner not unlike the `od` command, with a hex dump of the message contents, alongside an ASCII dump of it.

Only services with a published interface (specified in [brackets]) can be invoked. Not all services will blindly lend themselves to this type of invocation: Depending on the security policy, which is implemented differently by individual services, your service call request may be denied. If that is the case, the output of `service call` will contain a unicode error message, like so:

Output 9/1-6: Error messages returned from `service call`

```
# Attempt to call cancelMissedCallsNotification(), which requires
# MODIFY_PHONE_STATE permission
shell@htc_m8wl:/ $ service call phone 13
Result: Parcel(
  0x00000000: ffffffff 00000050 0065004e 00740069 '....P...N.e.i.t.'
  0x00000010: 00650068 00200072 00730075 00720065 'h.e.r. .u.s.e.r.'
  0x00000020: 00320020 00300030 00200030 006f006e ' .2.0.0.0. .n.o.'
  0x00000030: 00200072 00750063 00720072 006e0065 'r. .c.u.r.r.e.n.'
  0x00000040: 00200074 00720070 0063006f 00730065 't. .p.r.o.c.e.s.'
  0x00000050: 00200073 00610068 00200073 006e0061 's. .h.a.s. .a.n.'
  0x00000060: 00720064 0069006f 002e0064 00650070 'd.r.o.i.d...p.e.'
  0x00000070: 006d0072 00730069 00690073 006e006f 'r.m.i.s.s.i.o.n.'
  0x00000080: 004d002e 0044004f 00460049 005f0059 '..M.O.D.I.F.Y._.'
  0x00000090: 00480050 004e004f 005f0045 00540053 'P.H.O.N.E._.S.T.'
  0x000000a0: 00540041 002e0045 00000000          'A.T.E.......    ')
```

Once you get past permissions, however, (for example, by running as root), the possibilities of using `service call` in this manner are nearly endless, spanning all the features and capabilities of the Android frameworks. As we cover the framework services in this work one by one, we'll be showing their respective AIDL definitions, and number the calls accordingly.

2. The Binder

The discussion so far has mentioned the Binder several times, but kept it a very high level overview. Indeed, at a high level, suffice it to consider the Binder as a special type of a file descriptor, which - through a dedicated kernel driver - is connected to the service. This is also how Linux sees it, when the process is viewed through the `/proc/pid/fd` directory. Virtually every process in the system (With the exception of a few native processes) opens a handle to `/dev/binder`.

Much of Binder's inner workings, however, are shrouded in darkness - probably because, for most developers, ignorance is bliss. For those who want to know the details, there is, after all, always the source. For the scope of this work, however, it's beneficial to elucidate some of these dark corners and provide a closer view of Binder, explaining its functionality without going into the (not so well documented) source.

2.1. A little history

The Android Binder mechanism traces its root back to the Binder of another mobile operating system, BeOS. Binder served as the underlying support interconnecting BeOS's rich set of frameworks. Once heralded as the "next generation operating system", BeOS never gained much traction save for a few fans, and was eventually acquired by Palm. If the name doesn't ring a bell, that's fine - Palm Pilots were all the rage back at the end of the last millenium, catapulting 3COM to great heights before Palm was split off and spiraled back to earth. Palm was eventually acquired by HP, and its OS served as the basis for "WebOS", another venture that fell far short of its promise.

Binder, however, survived. Besides being ported to PalmOS (and integrated into their Cobalt architecture), it was also ported to other operating systems - including, of course, Linux. The Linux port was open sourced (at http://openbinder.org/, and though the website seems to have died since, some mirrors[1] survived). The original developers left Palm to join Android, and brought Binder with them. Chief amongst them was Dianne Hackborn, a well renowned developer and still one of the major figures driving Android today. An interview she gave to OSNews[2] back in 2006 explained the fundamentals of OpenBinder.

Android's implementation of Binder is more specific than OpenBinder, and - just like as originally intended in BeOS - serves as the fulcrum for all of its frameworks.

2.2. So, what, exactly, is Binder?

Binder is a Remote Procedure Call mechanism, allowing applications to communicate programmatically, but without having to worry about how to send and receive messages. From the application's perspective - server or client - all it needs to do is either call a method (client) or provide a method (service). When the client calls the method, the corresponding method is magically invoked in the service, with all the "details" handled transparently by Binder. These "minutiae" include:

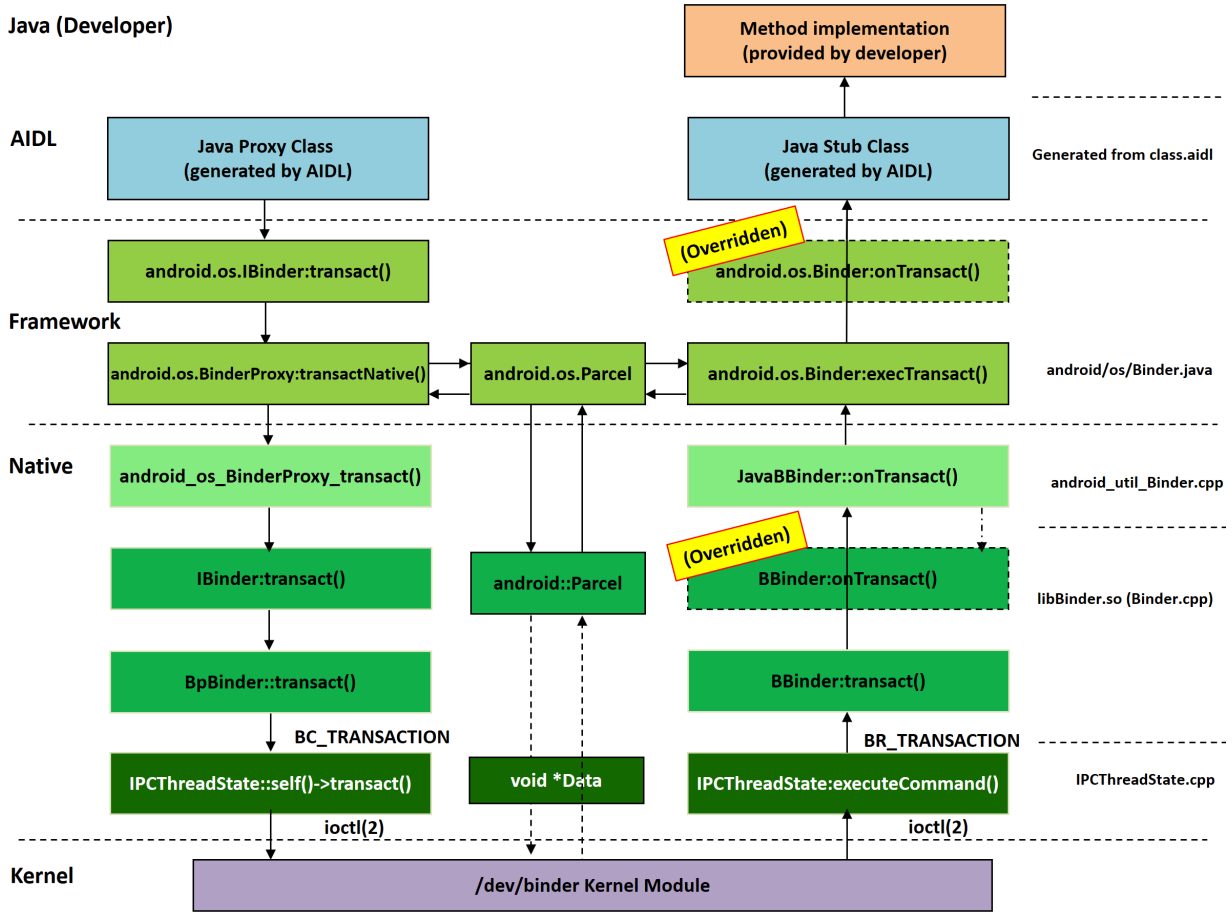
- **Locating the service process:** In most cases, the client and the service are two different processes (`system_server` notwithstanding). Binder needs to locate the service process for the client, so as to be able to deliver the message. This "location service" (also known as "endpoint mapping") is technically handled by `servicemanager`, as explained previously, but the `servicemanager` is only responsible for maintaining the service directory, mapping an interface name to a Binder handle. The "handle" is an opaque identifier, which was given to the `servicemanager` by Binder, and which only Binder knows the "true" meaning of - that is, the underlying PID wherein the service is located.
- **Delivering the message:** As we've seen, AIDL is used to generate the code which takes the parameters of the called method and serializes them (i.e. packs them into a structure in memory), or deserializes them (unpacks the structure back to individual parameters). The passing of the serialized structure from one process to another, however, is handled by Binder itself. Clients call the `BINDER_WRITE_READ ioctl(2)`, which sends the message over Binder, and blocks until a reply is returned (hence, the code - first write, then read).
- **Delivering objects:** The service handles mentioned previously are one type of an object Binder can pass, but so are **file descriptors** (just like UNIX Domain sockets). Passing around descriptors is an especially important feature, as it allows a trusted process (such as `system_server`) to natively open a device or socket for an untrusted process (such as a user app) - assuming the untrusted process has the required permission (as specified in the App's manifest).

- **Supporting credentials:** Inter process communication naturally has significant security aspects. A recipient of a message has to be able to verify the identity of the sender, so as not to be tricked into compromising overall system security. Binder is aware of its users' credentials - PID and UID - and securely embeds them in messages, so peers can operate with a reasonable level of security.
- **Death notifications:** When a Binder object or service dies (e.g. its process gets killed, the object is freed or the service is terminated), the Binder driver is able to detect this, and inform whomever has expressed interest - service peers or object holders - of the event via a "death notification" (informally, an "Obituary"). This notification enables the interested party to handle this condition, for example by retrying the connection or propagating an error.

2.3. Using Binder

Binder is used in all applications, whether or not the developers themselves realize it. The code involved in binder operates on many levels, as shown in Figure 9/2-1:

Figure 9/2-1: Message flow between client and server using Binder



All communication with the driver is performed through a single system call - `ioctl(2)` - with a set of `BINDER_*` codes. Chief amongst these is `BINDER_WRITE_READ`, which is called thus to emphasize the transactional nature of the facility: The caller sets both write and read buffers together, and subjects them to the driver, by which they will be consumed and populated (respectively). The write buffer to the driver consists of outgoing `BC_*` command codes. Similarly, the read buffer is filled with `BR_*` requests. The commands and replies mostly match eachother, so that a client's `BC_TRANSACTION` is delivered to the server as a `BR_TRANSACTION`, and the `BC_REPLY` from the server is returned as a `BR_REPLY` to the client.

In an effort to be true to the power user's view adopted in this work, this is as far as the discussion will go - for now. More detail on the various levels - from the Java objects, through AIDL, native, and kernel - can be found in Volume II.

2.4. 8.0+: The vndbinder and hwbinder

One of Project Treble's most notable changes was the re-structuring of Binder RPC into three different "namespaces", to enforce strict isolation between classes of AOSP and non-AOSP code:

- The /dev/binder node holds the "traditional" or "system" Binder namespace, which is reserved for AOSP servers and third party clients only. This means that most system and developer code continues to function in the same way, with no modification.
- The /dev/vndbinder node is a new node for vendor code, allowing vendor daemons to interact freely in and amongst themselves.
- The /dev/hwbinder node is another new node, for "Binderized-HAL" servers, as shown earlier in 1/3-4.

The `system_server` is, of course, able to use all three nodes, and thus can communicate with vendor code. The communication, however, must be initiated from the AOSP side, as vendor code - being barred from /dev/binder by SELinux policies - cannot initiate any contact with AOSP. This greatly improves system security through tight compartmentalization.

2.5. Tracing Binder

The Binder driver can multiplex any number of service connections over the same file descriptor. This means that a process will hold the character device descriptor irrespective of whether it is connected to one service, or to many. Indeed, a process can hold this descriptor and not be connected (yet) to any services at all. It follows, therefore, that there's no simple way to see exactly which services a given handle is connected to. If the Binder debug functionality is enabled through the Linux debugfs filesystem (/sys/kernel/debug/binder), however, Binder will emit debug data for every process, with entries for both node it owns, as well those it references. Each process using binder has a pseudo-file containing various statistics, and the `node` entries contained therein reveal the PIDs connected on the other end.

The `bindump` tool, which you can find on the Book's companion website can process this data and figure out who is connected to whom:

Output 9/2-2: Revealing binder endpoints using the `bindump` utility

```
# Get all services (copious output)
#
flame:/# /data/local/tmp/bindump users all
Service 'DockObserver' is node 14209
   Owner:    1169 (system_server)
   User:      604 (/system/bin/servicemanager)
Service 'SurfaceFlinger' is node 524
   User:    3618 (com.google.android.googlequicksearchbox:search)
    ....
   User:    1169 (system_server)
   User:     604 (/system/bin/servicemanager)
   Owner:    674 (/system/bin/surfaceflinger)
        ..
#
# Use with 'vnd' or 'hw' for other Binder namespaces:
#
flame:/# /data/local/tmp/bindump vnd users android.hardware.citadel.ICitadeld
Service 'android.hardware.citadel.ICitadeld' is node 8
   User:    1408 (/vendor/bin/hw/android.hardware.biometrics.face@1.0-service.google)
    ...
   Owner:    610 (/vendor/bin/hw/citadeld)
   User:     606 (/vendor/bin/vndservicemanager)
#
# Query single process handled (e.g. surfaceflinger)
#
flame:/# /data/local/tmp/bindump 674
Process 674 (/system/bin/surfaceflinger):
   Server: 'SurfaceFlinger' (node 524)
   Client: 'android.hardware.power.IPower/default' (node 110)
    ....
   Client: 'window' (node 11585)
```

Note, the Binder debug data has become restricted (thus requiring `root`) as of around 7.1. Further - in some devices (e.g. XiaoMi Mi 11) /sys/kernel/debug may not be mounted, leaving no way of figuring out Binder handles.

3. The servicemanager

By now we have seen that servicemanager forms the crux of Binder - without it, no client can find no server. This is indeed reflected in the servicemanager.rc from /system/etc/init:

Listing 9/3-1: The servicemanager definition, from /system/etc/init/servicemanager.rc

```
service servicemanager /system/bin/servicemanager
    class core animation
    user system
    group system readproc
    critical
    onrestart restart healthd
    onrestart restart zygote
    onrestart restart audioserver
    onrestart restart media
    onrestart restart surfaceflinger
    onrestart restart inputflinger
    onrestart restart drm
    onrestart restart cameraserver
    onrestart restart keystore
    onrestart restart gatekeeperd
    onrestart restart thermalservice
    writepid /dev/cpuset/system-background/tasks
    shutdown critical
```

What immediately stands out is just how many other key services are dependent on it, and must be restarted with it, in the event of a crash. Further, servicemanager is designated as critical, which means that init will aggressively attempt to restart it, or boot to recovery if it fails to do so after four successive attempts.

If any application or system component needs to use any other service, it must first consult the servicemanager to obtain a handle. Similarly, services cannot expect clients until they register their presence with it. It is for this reason that, if the manager is restarted, so must all of its dependents - after all, restarting implies the service directory must be rebuilt from scratch, and services thus need to register. It likewise follows that, if servicemanager cannot operate, Inter-Process Communication (IPC) cannot subsist. The servicemanager holds handles to all services registered with it (so it can dole them out to requestors), and is therefore technically a client of all of them (as can be seen in Output 9/2-2).

The servicemanager is a small, single-threaded binary, with a simple operation. Up until Android 11, a call to binder_open() obtains the /dev/binder descriptor. This is followed by a call to binder_become_context_manager(), to establishes its role as an endpoint locator. Thereafter, the servicemanager enters an endless binder_loop, which blocks on the descriptor, until a transaction (i.e. request from a client) occurs. This wakes the process, and calls its svcmgr_handler() callback, which processes the transaction. The flow at this point is usually one of two - addService or getService - and both paths are illustrated in Figure 9/3-2.

The service lookup must somehow be bootstrapped - in other words, the servicemanager should be globally accessible, so that services can register with it, and clients can look them up. At the native level, services and clients alike can call on defaultServiceManager() to get a handle to the service manager (technically, to its interface, as a sp<IServiceManager>). The interface (defined in IServiceManager.h with no official AIDL) exposes a simple set of four transaction request codes - [GET/CHECK]_SERVICE (1,2), ADD_SERVICE (3) and LIST_SERVICES (4). Up until Android 11 there is no API to remove a service. Services are automatically removed when their processes die, because Binder can detect that, and send a death notification.

3.1. The android.os.IServiceManager Interface (Android 11)

manager	
Interface:	android.os.IServiceManager
Manager (Proxy):	None
Implementation:	ServiceManager.cpp
Servers:	None (self)
Clients:	Everyone

Android 11's servicemanager implementation has been rewritten in C++, and is considerably more complex, although the general flow is still roughly the same. The main() uses libbinder's ProcessState singleton to initialize the device handle, and adds itself as a service to an internal ServiceManager object. It then calls ProcessState's becomeContextManager() method, and enters an endless Looper, with a BinderCallback object adding the device file descriptor to the looper, and setting a handleEvent callback.

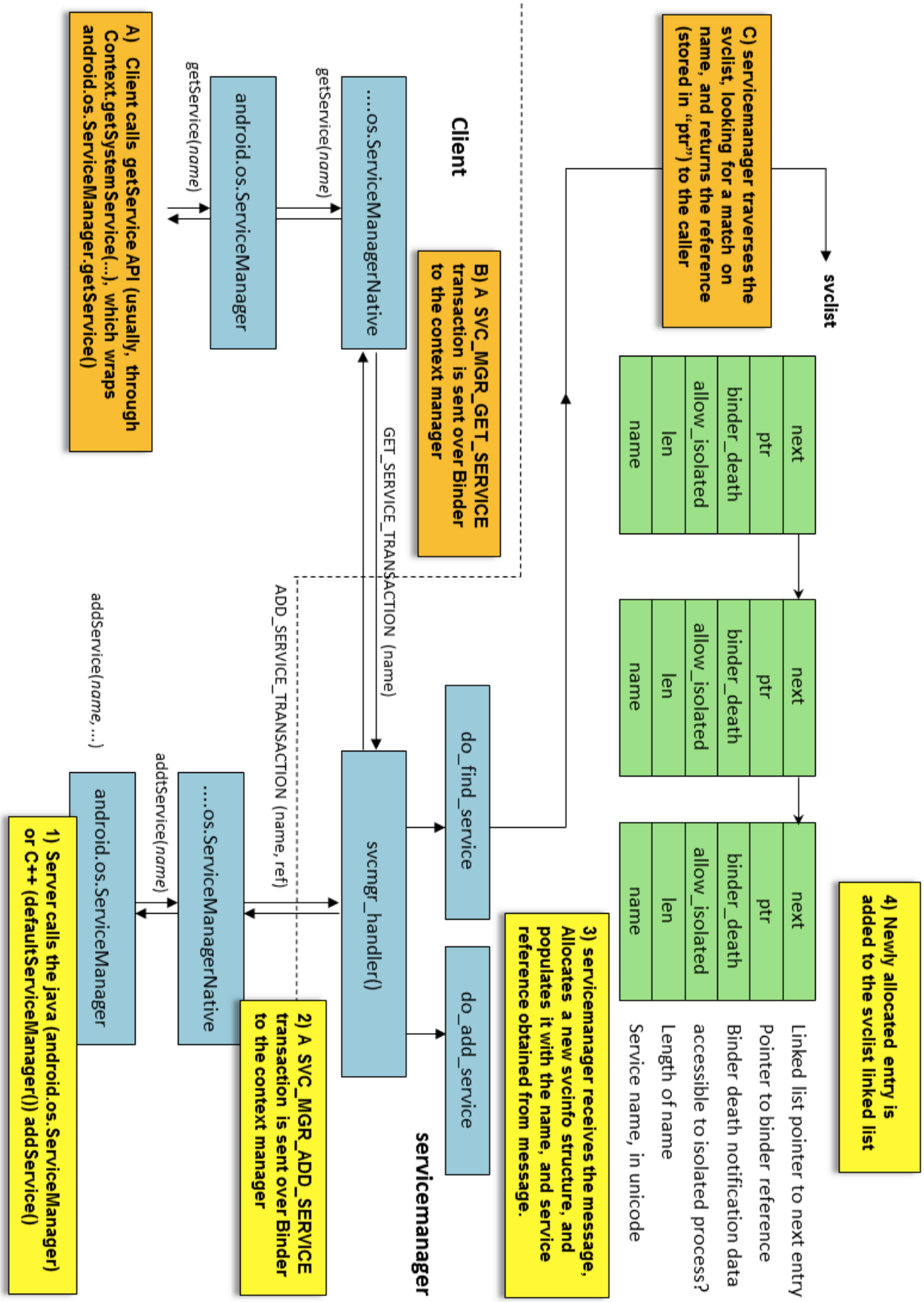


Figure 9/3-2: Registering and accessing Android framework services

The changes allow `servicemanager` to provide callbacks and nontifications to clients. The `ServiceManager` object implements the `android.os.IServiceManager` interface. This interface, although having been implicit up to that point, has been formally defined (in frameworks/native/libs/binder/aidl/android/os/IServiceManager.aidl), and significantly extended: New methods have been added past the traditional four, in order to support the notifications and client callbacks, as well as service de-registration. The AIDL also supports `init`'s `interface` directive (as of 9.0), which allows the "dynamic" start of a service by `servicemanager` when a service lookup fails, by having `servicemanager` set the `ctl.interface_start` property (see 8/2.2).

Table 9/3-3: methods exported by the `android.os.IServiceManager` interface

#	API	Notes
1	**getService**(*name, &outBinder*)	Get a handle to the service specified by *name*
2	**checkService**(*name, &outBinder*)	
3	**addService**(*name, &binder allowIsolated, dumpPriority*)	Used by servers to register themselves with the service manager. Servers can decide whether or not they want to allow isolated (sandboxed) processes to connect
4	**listServices**(*&dudmpPriority, outList*)	Return a vector (list) of all services. Not used by the framework, but used by `service list`
5,6	**[un]registerForNotifications**(*name, &callback*)	Register an *IServiceCallback* handler for *name* notifications
7	**isDeclared**(*name, &outReturn*)	Return boolean *outReturn* if *name* is declared
8	**registerClientCallback**(*name, binder, &callback*)	Register an *IClientCallback* handler for *name* notifications
9	**tryUnregisterService**(*name, binder*)	Attempt to deregister *binder* handle for *name*

The programmatic APIs are wrapped by the framework class `a.os.ServiceManagerNative`, which is further encapsulated in `android.os.ServiceManager`. Apps aren't expected to use this directly, and instead call on `Context.getSystemService()` in order to look up system services, and use intents for third party services. Either way, registration and lookup of services - both system and third party - is performed over Binder, as shown in the previous figure.

Both registering and looking up services are considered security-sensitive operations: Apps (i.e. processes with UID >= `AID_APP`) are explicitly disallowed from adding, and isolated apps ((UID in `AID_ISOLATED_[START/END]` cannot lookup services unless those have explicitly requested `allowIsolated`. A low-level `selinux_check_access()` call (wrapped in Android 11 by `Access::actionAllowedFromLookup()` and before that by `check_mac_perms()`) queries the SELinux policy. The policy differentiates between system (`plat`), `vendor` and `product` service and hwservice contexts, based on files in …/etc/selinux. A discussion of SELinux is left for Volume III.

Experiment: Using the `service` command to interface with service manager

Android provides the `service` command line utility as a simple interface for the service manager. This simple utility also demonstrates how to use the programmatic APIs to query services. Using `service list` you can display all registered services, as well as their published interfaces (discussed later in this chapter), and using `service check`, see if a given service can be contacted. Output 9/3-4 shows the service lookup from `service list`, focusing on the first service handle returned:

Output 9/3-4: a `jtrace` of `servicemanager` responding to `service check power`

```
flame:/ $ /data/local/tmp/jtrace64 /system/bin/service list
# Binder library initialization: Verify version and set thread pool
#
ioctl (3 </dev/binder>, BINDER_VERSION, 0x7fe5e55664 - 0x43a0ecfa00000000
ioctl (3 </dev/binder>, BINDER_SET_MAX_THREADS, 0x7fe5e55658 - 15)
..
#
# PING_TRANSACTION (_PNG) to servicemanager, to ensure connectivity
#
ioctl (3 </dev/binder>, BINDER_WRITE_READ (0xc0306201):
   Request (68/68 bytes @0xb400007ccd835850):
        0x00: 0x40406300 BC_TRANSACTION on DefaultServiceManager, Code '_PNG'
        Method:  ::PING_TRANSACTION
   Reply (76/76 bytes @0xb400007ccd8342f0):
        0x00: 0x720c BR_NOOP
        0x04: BR_TRANSACTION_COMPLETE (0x7206)
        0x08: 0x80407203 BR_REPLY: (0)
#
# LIST_SERVICES(....) returns a buffer with count of services (200) and array of names
#
ioctl (3 </dev/binder>, BINDER_WRITE_READ (0xc0306201):
    Request (96/96 bytes @0xb400007ccd835850):
        0x00: 0x40086303 BC_FREE_BUFFER @7b9d6c3000
        0x0c: 0x40046304 BC_INCREFS for target 0x0
        0x14: 0x40046305 BC_ACQUIRE for target 0x0
        0x1c: 0x40406300 BC_TRANSACTION on DefaultServiceManager, Code 4
        Method: android.os.IServiceManager::LIST_SERVICES(15)

    Reply (76/76 bytes @0xb400007ccd8342f0):
        0x00: 0x720c BR_NOOP
        0x04: BR_TRANSACTION_COMPLETE (0x7206)
        0x08: 0x80407203 BR_REPLY: (200, "DockObserver", "SurfaceFlinger", ....) ...

writev  (1 </dev/pts/1>,"Found 200 services:\n",1) = 20
..
#
# Request handle for SurfaceFlinger, shown here because DockObserver has no interface)
#
ioctl (3 </dev/binder>, BINDER_WRITE_READ (0xc0306201):
    Request (96/96 bytes @0xb400007ccd835850):
        0x00: 0x40086303 BC_FREE_BUFFER @7b9d6c3000
        0x0c: BC_RELEASE for target 0x1
        0x14: 0x40046307 BC_DECREFS for target 0x1
        0x1c: 0x40406300 BC_TRANSACTION on DefaultServiceManager, Code 3
        Method: android.os.IServiceManager::CHECK_SERVICE(SurfaceFlinger)

    Reply (76/76 bytes @0xb400007ccd8342f0):
        0x00: 0x720c BR_NOOP
        0x04: BR_TRANSACTION_COMPLETE (0x7206)
        0x08: 0x80407203 BR_REPLY: (sh* to handle 1 PID 667, /system/bin/surfaceflinger)
#
# INTERFACE_TRANSACTION (_NTF) to SurfaceFlinger process:
#
ioctl (3 </dev/binder>, BINDER_WRITE_READ (0xc0306201):
    Request (96/96 bytes @0xb400007ccd835850):
        0x00: 0x40046304 BC_INCREFS for target 0x1
        0x08: 0x40046305 BC_ACQUIRE for target 0x1
        0x10: 0x40086303 BC_FREE_BUFFER @7b9d6c3000
        0x1c: 0x40406300 BC_TRANSACTION on target 0x1, Code '_NTF'
        Method: ::INTERFACE_TRANSACTION

    Reply (76/76 bytes @0xb400007ccd8342f0):
        0x00: 0x720c BR_NOOP
        0x04: BR_TRANSACTION_COMPLETE (0x7206)
        0x08: 0x80407203 BR_REPLY: "android.ui.ISurfaceComposer"
writev  (1 </dev/pts/1>,"1\tSurfaceFlinger: [android.ui.ISurfaceComposer]\n",1) = 48
1       SurfaceFlinger: [android.ui.ISurfaceComposer]
```

4. system_server

Android devices have dozens of services, and as more are added with each version, this number approaches two hundred in Android 11 - before even considering vendor services. Fortunately, the vast majority of framework services are simple enough that they do not require their own process, and can instead run as threads. These threads, however, need a host process to run in - and that is exactly what `system_server` provides. Note, however, some services do not have dedicated threads. It is also important that not all the services are visible to applications: Some, like the `Installer` are internal, and thus invisible both to apps as well as `service list`, as they cannot be accessed over Binder, and are only visible through their objects in `system_server`'s namespace.

Similar to Windows' `svchost.exe`, the `system_server` provides little more than a shell - a container process. The two can also be compared in the sense that svchost.exe loads services through dynamically linked libraries (DLLs), whereas `system_server` loads Java classes. In Android, however, this is even more important a function: Though the Dalvik VM is optimized for sharing, running services alongside one another in the same VM provides an even greater savings in resources. This does not come without a bit of risk, however, as a misbehaving service can thus affect its siblings. For the most part, though, this isn't much of a concern, as only Android's system services, and not those of the vendor or additional apps, are allowed to run inside `system_server`.

The `system_server` is not a native app: It is implemented mostly in Java, with some JNI calls. The services it loads are similarly implemented in Java, though a great deal of them also rely on JNI to escape the virtual machine and interact with hardware components. The `zygote` automatically starts `system_server` when it itself is started by the /init.rc (q.v. Listing 8/6-1) with the `--start-system-server` switch. The switch makes zygote invoke `startSystemServer()`, in which are hardcoded the arguments - capabilities, group memberships (`--setgroups`), the "nice name" (`system_server`), and the class to load - `com.android.server.SystemServer`.

The `system_server` **does not** execute with root privileges, but comes pretty close - UID:GID of `system:system`, enhanced capabilities, and a host of secondary group memberships, which enable it to access hardware devices and perform privileged operations.

4.1. Handling Services

Services are internally represented as `com.android.server.SystemService` instances. The abstract class provides the service lifecycle methods:

- **`onStart()`**: Signalling service start. The service is expected to perform whatever initialization steps are necessary (e.g. register handlers, receivers, etc.), and publish whichever interfaces other system components can use to interact with it. Interfaces are primarily Binder endpoints (using `publishBinderService(…)`, to register with the `servicemanager`), but can also be local (using `publishLocalService()`, which adds the service to the `LocalServices` static, visible only within `system_server`). This method remains `abstract`, and so must be implemented when extending the class.
- **`onBootPhase(`*`phase`*`)`**: Signalling one of several "boot phases", allowing the service to adapt its behavior based on the system's maturity, availability of other components, or SafeMode, though the default implementation does nothing. The boot phases have numeric values and are defined in increasing temporal order, shown in Table 9/4-1:

Table 9/4-1: The boot phases defined in `c.a.server.SystemServiceManager`

###	PHASE_..	Description
100	`..WAIT_FOR_DEFAULT_DISPLAY`	Earliest stage, display not ready
480	`..LOCK_SETTINGS_READY`	Lock settings service is available
500	`..SYSTEM_SERVICES_READY`	Core AOSP system services are ready and safe to use
520	`..DEVICE_SPECIFIC_SERVICES_READY`	Device/vendor specific services are ready and safe to use
550	`..ACTIVITY_MANAGER_READY`	`ActivityManager` is ready, intents/broadcasts safe to use
600	`..THIRD_PARTY_APPS_CAN_START`	Apps are available to both call and be called at this stage
1000	`..BOOT_COMPLETED`	Home application started, full UI available

- **onUser[Starting/Stopping/Stopped/Switching/Unlocking](*targetUser*), etc**: allowing services to respond to the (human) user lifecycle, in multi-user environments where different people use the same device. These are optional, and the service can choose to override `isUserSupported(targetUser)` according to the types of users supported.

The `SystemService` objects are managed through a `SystemServiceManager` instance. The object's most important method is `startService(Class[Name])`: When passed a *className* or *class* extending `com.android.server.SystemService`, `startService(…)` uses reflection to construct an instance of the service class, add to an internal `mServices` array, and then call the instance's `onStart()` method. The `SystemServiceManager` is also the one to signal boot phases (through `StartBootPhase(…)` method), and drive the service user callbacks.

4.2. Startup and Flow

For such an important fulcrum of the entire system, `system_server` has a rather simple flow. Once it has forked off from `zygote`, the child process drops its privileges, and toggles the capabilities as discussed above. It then proceeds to load the class, whose `static main()` creates an instance and calls `run()`. Instantiation records start timestamps, and checks if the system is being restarted through the `sys.boot_completed` property. Another important check is for "factory mode" through `FactoryTest.getMode()`, which inspects the `ro.factorytest` property, shown in Table 9/4-2. The `run()` method continues with the full initialization flow, depicted in Figure 9/4-3 (opposite page).

Table 9/4-2: Factory test values `ro.factorytest`) and their impact on startup

value	#define	Implies
0 (default)	`FACTORY_TEST_OFF`	Normal startup.
1	`FACTORY_TEST_HIGH_LEVEL`	
2	`FACTORY_TEST_LOW_LEVEL`	No bluetooth, input, accessibility, lock settings

There are numerous system services to start, and `system_server` needs to instantiate them one by one. Android 5.0 started refactoring this flow, and the effort was completed by 8.0: The flow is significantly simplified by grouping services of similar classification into three "classes":

- **Bootstrap services**: These include the `PlatformCompat[Native]`, `FileIntegrity`, `Installer`, `DeviceIdentifiersPolicy`, `UriGrantsManager`, `ActivityManager`, `DataLoaderManager`, `Incremental`, `PowerManager`, `ThermalManager`, `RecoverySystem`, `Lights`, `Sidekick` (Android Wear), `DisplayManager`, `PackageManager`, `OtaDexopt`, `UserManager`, `AttributeCache`, `OverlayManager`, and `SensorPrivacy`. Additionally, a check is performed if the device's `/data` partition is encrypted or in the process of encryption - which affects startup by starting only apps designated as "core apps", though these have no relation to the next class.
- **Core services**: These include the `SystemConfigService`, `BatteryService`, `UsageStatsService`, `WebViewUpdateService` (if `FEATURE_WEBVIEW`), `CachedDeviceStateService`, `BinderCallsStatsService`, `LooperStatsService`, `RollBack`, `BugReport` and `GpuService`.
- **"Other" services**: basically, everything else. There are dozens of services in this class (which the source admits is "a miscellaneous grab bag of stuff that has yet to be refactored and organized"). Android Wear (detected through `FEATURE_WATCH`) adds about a half dozen services here as well. It is during this stage that all boot phases in Table 9/4-1 are signalled.

Most services are started synchronously, but some services (such as the sensor services, Android 11's Blob store and the HIDL services) are started asynchronously through the `SystemServerInitThreadPool`. One way or another, once all services are started, `SystemServer`'s startup is complete. The main thread therefore enters its looper, which hopefully loops indefinitely. We say "hopefully", since the looper is not expected to exit, and will throw a runtime exception if it does. Internally, the loop blocks, polling its file descriptors (and in particular, its Binder handle) for incoming messages. When messages arrive, they are dispatched to their respective targets.

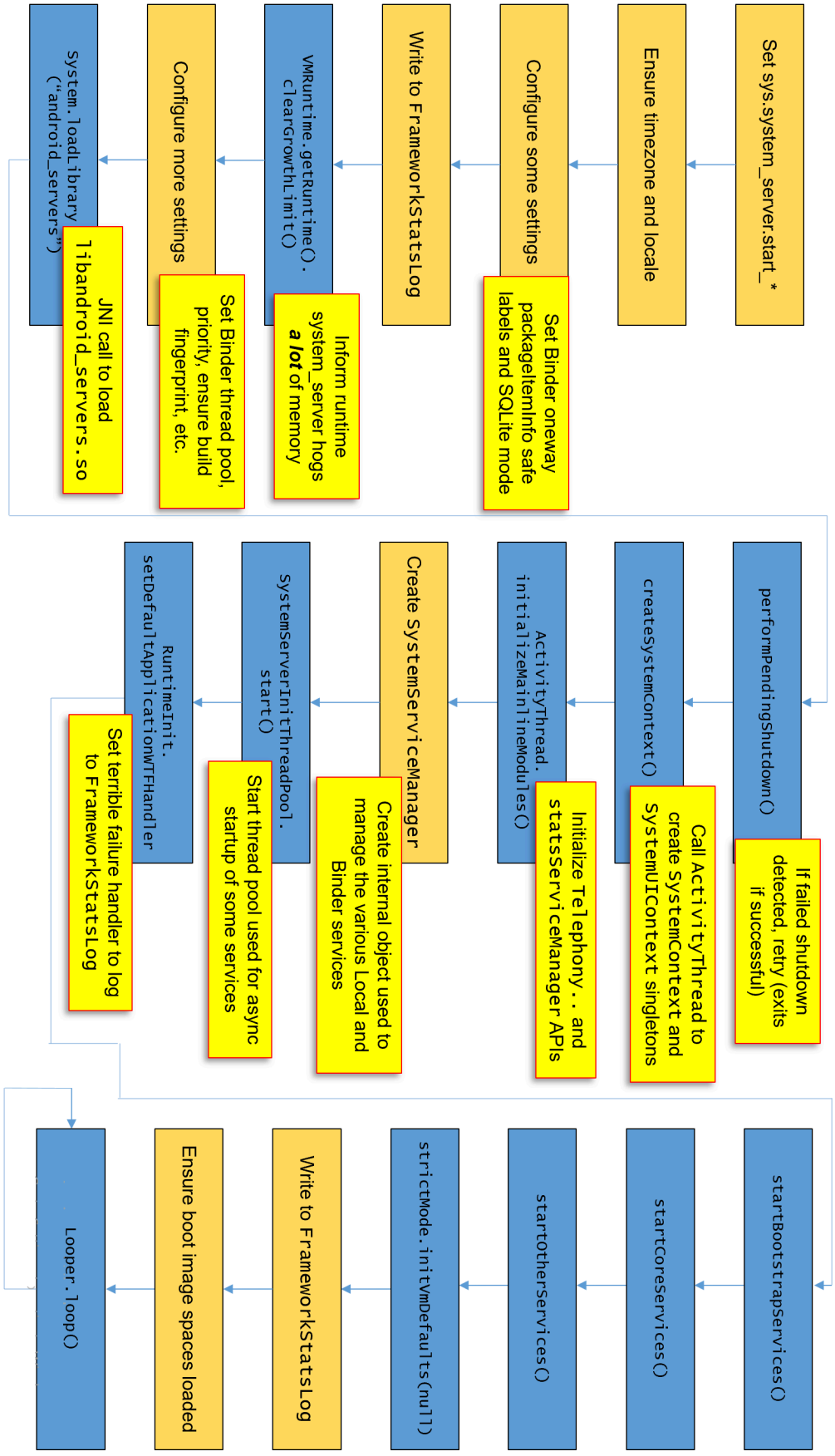


Figure 9/4-3: The startup flow of `system_server` (`com.android.server.SystemServer::run()`)

Experiment: Unraveling the threads of `system_server`

Linux thread objects may be named when created. Naming a thread calls the underlying `prctl(2)` system call - a little known but highly useful API which allows the renaming of threads and processes at the kernel level. The name is then visible through the /proc filesystem in the status proc entry of the thread. The method is not perfect, as it allows for only 16 characters in a name - but it sure beats rummaging through random thread identifiers, trying to figure out which does what. `system_server`'s threads are almost all created by Java, whose `Thread` class gets a name argument, as do the Android `HandlerThread`s, `ServiceThread`s etc.

Using a basic command pipeline you can easily enumerate the threads, and get their individual names (this works on any process, so as long as the `for` iterates over its task/ subdirectory, which contains a directory entry for each thread). Binder threads and thread pools are omitted from this output, which has also been edited to allow annotations and group together threads from the same subsystems. Note, that while TIDs aren't normally predictable, a large part of system_server's are started incrementally, and so looking at the IDs can give you a sense as to the system's framework startup (race conditions notwithstanding).

Output 9/4-4: Iterating through threads

```
flame:/proc/1169/task # grep Name */status | grep -v Binder | grep -v pool- | grep -v Thread-
#
# Dalvik/ART maintenance threads
#
1176/status:Name:   Signal Catcher
1177/status:Name:   perfetto_hprof_
1178/status:Name:   Jit thread pool
1179/status:Name:   HeapTaskDaemon
1180/status:Name:   ReferenceQueueD
1181/status:Name:   FinalizerDaemon
1182/status:Name:   FinalizerWatchd
#
# ServiceThread subclasses (exc. watchdog)
#
1185/status:Name:   android.fg        # c.a.s.FgThread
1186/status:Name:   android.ui        # c.a.s.UiThread
1187/status:Name:   android.io        # c.a.s.IoThread
1188/status:Name:   android.display   # c.a.s.DisplayThread
1189/status:Name:   android.anim      # c.a.s.AnimationThread
1190/status:Name:   android.anim.lf   # c.a.s.wm.SurfaceAnimationThread
1191/status:Name:   watchdog          # c.a.s.Watchdog
1193/status:Name:   android.bg        # c.a.internal.os.BackgroundThread
#
# Services begin with ActivityManager, spawning multiple workers and sub-threads
#
1194/status:Name:   ActivityManager
1195/status:Name:   ActivityManager
1196/status:Name:   ActivityManager
1197/status:Name:   ActivityManager
1199/status:Name:   OomAdjuster       # ActivityManager$OomAdjuster
1200/status:Name:   batterystats-wo   # a.s.am.BatteryExternalStatsWorker
1202/status:Name:   FileObserver      # FileObserver$Thread
1203/status:Name:   CpuTracker        # Created by ActivityManager
1567/status:Name:   CachedAppOptimi   # 11.0: c.a.s.am.CachedAppOptimizer
1707/status:Name:   TaskSnapshotPer   # c.a.s.wm.TaskSnapshotPersister
3078/status:Name:   LazyTaskWriterT   # c.a.s.wm.PersisterQueue
#
# Miscellaneous
#
1207/status:Name:   PowerManagerSer   # Created by PowerManagerService
1208/status:Name:   BatteryStats_wa   # c.a.s.am.BatteryStatsService wakeupReason
1209/status:Name:   PackageManager    # PackageManager
1315/status:Name:   PackageInstalle   # Created by PackageManager
1528/status:Name:   SensorEventAckR   # f/n/s/sensorservice/SensorService.cpp
1530/status:Name:   SensorService     # f/n/s/sensorservice/SensorService.cpp
1531/status:Name:   HealthServiceHw   # c.a.s.BatteryService HandlerThread
1754/status:Name:   HealthServiceHw   # c.a.s.BatteryService HandlerThread (2)
1534/status:Name:   RollbackPackage   # c.a.s.rollback.RollbackPackageHealthObserver
1535/status:Name:   RollbackManager   # c.a.s.rollback.RollbackManagerServiceImpl
1541/status:Name:   AccountManagerS   # c.a.s.accounts.AccountManagerService.java
1550/status:Name:   SettingsProvide   # c.a.providers.settings.SettingsProvider
1581/status:Name:   AlarmManager      # c.a.s.AlarmManagerService
#
# Input Flinger subsystem (q.v. II/9)
#
1599/status:Name:   InputDispatcher   # f/n/s/inputflinger/dispatcher/InputDispatcher.cpp
1600/status:Name:   InputReader       # f/n/s/inputflinger/reader/InputReader.cpp
2078/status:Name:   InputClassifier   # f/n/s/inputflinger/InputClassifier.cpp
```

Experiment (cont.): Unraveling the threads of `system_server`

Output 9/4-4 (cont.): Iterating through threads

```
#
# Connectivity and networking:
#
1673/status:Name:   ConnectivitySer   # Created by ConnectivityManager
1674/status:Name:   roid.pacmanager   # c.a.s.connectivity.PacManager
1692/status:Name:   ConnectivityThr   # a.net.ConnectivityThread

1675/status:Name:   NsdService        # c.a.s.NsdService (Neighbor Svcs Disc. state machine)
1676/status:Name:   mDnsConnector     # Created by NsdService
1602/status:Name:   NetworkWatchlis   # c.a.s.net.watchlist.NetworkWatchlistService
1858/status:Name:   NetworkStatsObs   # c.a.s.net.NEtworkStatsObservers
1642/status:Name:   NetworkStats
1643/status:Name:   NetworkPolicy     # a.server.net.NetworkPolicyManagerService
1644/status:Name:   tworkPolicy.uid   # a.server.net.NetworkPolicyManagerService

1607/status:Name:   hidl_ssvc_poll
1610/status:Name:   AppIntegrityMan   # 11.0 AppIntegrityManager
1616/status:Name:   StorageManagerS   # c.a.s.StorageManagerService
1631/status:Name:   LockSettingsSer
#
# WiFi subsystem threads (q.v. II/12)
#
1664/status:Name:   AsyncChannelHan   # c.a.s.wifi.WifiInjector
1665/status:Name:   WifiHandlerThre   # c.a.s.wifi.WifiInjector
1667/status:Name:   WifiP2pService    # c.a.s.wifi.WifiInjector
1668/status:Name:   PasspointProvis   # c.a.s.wifi.hotspot2.PasspointProvisioner
1672/status:Name:   WifiScanningSer   # c.a.s.wifi.scanner.WifiScanningService
1703/status:Name:   wifiRttService    # c.a.s.wifi.WifiInjector
1704/status:Name:   wifiAwareServic   # c.a.s.wifi.WifiInjector
1705/status:Name:   EthernetService
1706/status:Name:   WifiManagerThre   # a.n.wifi.WifiFrameworkInitializer
3137/status:Name:   RedirectListene   # c.a.s.wifi.hotspot2.PasspointProvisioner
3148/status:Name:   OsuServerHandle   # c.a.s.wifi.hotspot2.OsuServerConnection
#
# Notifications, etc.
#
1677/status:Name:   ranker            # c.a.s.notification.NotificationManagerService
1678/status:Name:   notification-sq   # c.a.s.notification.NotificationUsageStats
1679/status:Name:   onProviders.ECP   # c.a.s.notification.EventConditionProvider
1680/status:Name:   DeviceStorageMo   # DeviceStorageMonitorService handler thread
1681/status:Name:   AS.SfxWorker      # c.a.s.audio.SoundEffectsHelper
1683/status:Name:   AudioService      # Created by AudioService$AudioSystemThread
1684/status:Name:   AudioDeviceBrok   # c.a.s.audio.AudioDeviceBroker
1688/status:Name:   UEventObserver    # Kernel uevent observer (shared by many services)
1693/status:Name:   backup            # Created by BackupManagerService
1695/status:Name:   BlobStore         # 11.0 BlobStore
1696/status:Name:   GraphicsStats-d   # a.graphics.GraphicsStatsService (GraphicsStats-disk)
1698/status:Name:   SessionRecordTh   # c.a.s.media.MediaSessionService
1699/status:Name:   SliceManagerSer   # c.a.s.slice.SliceManagerService
1700/status:Name:   CameraService_p   # c.a.s.camera.CameraServiceProxy
1701/status:Name:   StatsCompanionS
1711/status:Name:   PhotonicModulat   # c.a.s.display.DisplayPowerState$PhotonicModulator
1774/status:Name:   SyncManager
1811/status:Name:   UsbService host
1917/status:Name:   EmergencyAfford   # c.a.s.emergency.EmergencyAffordanceService
1933/status:Name:   NetworkTimeUpda   # NetworkTimeUpdateService's HandlerThread
2008/status:Name:   CCodecWatchdog    # f/av/media/codec2/sfplugin/CCodec.cpp
2010/status:Name:   NDK MediaCodec_   # f/av/media/ndk/NdkMediaCodec.cpp
2338/status:Name:   BluetoothRouteM
#
# Telecom
#
2350/status:Name:   AudioPortEventH   # a.media.AudioPortEventHandler
2369/status:Name:   uteStateMachine   # c.a.s.telecom.CallAudioRouteStateMachine
2372/status:Name:   CallAudioModeSt   # c.a.s.telecom.CallAudioModeStateMachine
2373/status:Name:   ConnectionSvrFo   # c.a.s.telecom.ConnectionServiceFocusManager
2867/status:Name:   AdbDebuggingMan   # c.a.s.adb.AdbDebuggingManager
5131/status:Name:   AsyncQueryWorke
5542/status:Name:   GrallocUploadTh
```

5. A bird's eye view of framework Services

AOSP defines well over a hundred services, with the number growing closer to 200 by Android 11. Vendor-added services can increase this even further. While some are designed for use by applications, most are internal, and thus undocumented. Even the application-facing services, however, have some undocumented APIs. It's no surprise, then, that a signficant portion of this work needs to be devoted to providing a little bit more clarity as to their operation.

Android subdivides its services by package namespace. The following namespaces are used:

- **`com.android.internal`** - is used, as the name implies, for internal services, with subpackages for `telephony`, `app` and `appwidget`.
- **`android.net`** - is used for the various Wi-Fi and connectivity related services (though not telephony, as those are handled by `com.android.telephony`.
- **`android.app`** - groups together services used for application support .
- **`android.content`** - services loosely associated with Android content providers
- **`android.os`** - services used for operating system support, such as the `UserManager`, `PowerManager` and others.
- **`android.media`** - services associated with audio/video presentation and management.

Some namespaces, such `android.media`, indeed contain any and every service associated with their group. The classification gets blurry, however, with namespaces such as `android.os`, `android.app` and `android.content`, wherein services are grouped with little apparent connection. It's also not uncommon to see a service move in between packages from one Android release to another.

The approach taken by this work, then, aims to tackles services a little bit differently. As shown in Table 9/5-1 over the next several pages, services have been classified by functionality, rather than namespace:

Table 9/5-1: Android Services, categorized by functionality

Category	Service Name	Handles
Application (II/3)	activity	Activity Manager (manages lifecycle)
	activity_task	Activity Task Manager
	app_binding	10.0: Keep apps running
	app_prediction	Predict app/shortcut usage
	app_integrity	11.0: Package install verification
	appwidget	Widgets
	content	Content provider sync, observers, etc.
	content_capture	10.0: Content capture services
	content_suggestions	
	launcherapps	Application/Launcher interface
	notification	Notifications, Toasts etc
	slice	Application Slices
Application Debug	binder_call_stats	Binder statisticss (II/7)
	cacheinfo	11.0: Binder cache info
	dbinfo	SQLite database usage info
	looper_stats	Application looper statistics (II/3)
	runtime	Core Library Debug Info

Table 9/5-1: Android Services, categorized by functionality

Category	Service Name	Handles
Application Services (II/4)	alarm	Deferred execution
	autofill	Auto-fill input boxes
	blob_store	11.0: Shared datasets blob manager
	shortcut	Shortcut/deep linking
	backup	Application Backup agents
	clipboard	Clipboard (cut/paste) services
	jobscheduler	Deferred job execution
	print	Print to local or network printer
	search	Search using registered activity
	textclassification	Classify text and context for suggestions
	textservices	Spell check
	update_lock	Acquire lock before a system update
Device Configuration & Management (Chapter 10)	user	User management
	account	Account management
	crossprofileapps	Applications across profiles
	device_config	Device Configuration
	settings	User and device profile settings
	system_config	11.0: Interface to etc/sysconfig/... files
Diagnostics (Chapter 12)	bugreport	dumpstate
	dropbox	Persistent log/blob store service
	stats	Gather system-wide statistics
	stats_companion	
	incident	9.0: Incident reporting
	incident_companion	
Graphics (II/11)	SurfaceFlinger	Surface compositor
	window	Window manager
	display	Display management
	color_display	Color display
	gpu	GPU driver details
	graphicsstats	Graphic statistics
	gfxinfo	Graphics information for `dumpsys`
Hardware (Various)	DockObserver	Detects device "docking" over USB
	consumer_ir	Infra Red blasters
	vibrator	Device vibrator ("buzz" when in silent mode) and haptic feedback
	external_vibrator_service	
	device_identifiers	Get device serial number and/or other identifiers
	lights	A11: Led/light management
	serial	Serial device enumeration and access proxy
	sensorservice	Sensors
	sensor_privacy	Enable/disable sensor privacy
	nfc	Near Field Connectivity. Owned by `com.android.nfc`
	contexthub	Context Hub Nano App interface
	usb	Universal Serial Bus interface
	bluetooth_manager	BlueTooth Management
Input (II/9)	input_method	Input Method Editor (IME) support
	input	Input Manager
	inputflinger	Combine multiple input sources
I/O (Chapter 11)	iorapd	I/O Read Ahead and Pin Daemon
	pinner	Pins important files in memory

Table 9/5-1 (cont.): Android Services, categorized by functionality

Category	Service Name	Handles
Location (II/12)	country_detector	Detect country and locale
	location	Determine location from GPS, WiFi, Cell, etc
	network_time_update_service	Network time sync
	time_detector	Suggests time and time zone from manual, network or telephony time sources
	time_zone_detector	
Media (II/9)	audio	Audio Subsystem
	media.camera	Camera Services
	media.camera[.proxy]	
	media_projection	Project media (Miracast, virtual displays, etc.)
	media_router	Route media to different display/speakers
	media_session	Manage media sessions
	midi	Musical Instrument Digital Interface (MIDI)
	soundtrigger[_middleware]	Sound Trigger ("Ok Google")
	media.aaudio	Native audio stream control/notifications
	media.audio_policy	Audio Policy (volume, effects, etc)
	media.extractor	Codec extraction
	media.resource_manager	Manage & monitor client media resources
	media_resource_monitor	
	media.player	Media recording/playing sevicesw
	media.audio_flinger	Combines several audio-streams together
	media.metrics	Maintain audio/video metrics
	drm.drmManager	Digital Rights management
	media.drm	
Mobile Device Management (III)	device_policy	Device Policy Management (Admin apps)
	restrictions	11.0: Obtain package restrictions from provider
Networking (II/11)	network_stack	Network stack monitoring
	netd_listener	Owned by /system/bin/netde
	connmetrics	IP connectivity metrics/events
	servicediscovery	Neighbor Service Discovery (mDNS)
	connectivity	Query, monitor and change network state
	ethernet	Ethernet (wired) interface management
	netpolicy	Network policy restrictions/control
	netstats	Network Statistics
	network_score	Network score evaluator
	dnsresolver	Domain Name Server resolver service
	ipsec	IPSec encryption/authentication
	network_management	Network Management
	network_watchlist	Network traffic watchlist
	netd	Network Daemon
	wifip2p	WiFi Peer-to-Peer Management
	wifiaware	WiFi Aware (discovery/peer-to-peer data connections)
	wifirtt	802.11mc WiFi Round Trip Time
	wifiscanner	Network scanning
	wifi	General WiFi services
	wifinl80211	11.0: Wificond (Wi-Fi subsystem manamagent)
	tethering	11.0: Tether controller

Table 9/5-1 (cont.): Android Services, categorized by functionality

Category	Service Name	Handles
Process/Thread (Chapter 11)	cpuinfo	Process CPU utilization statistics
	meminfo	Memory utilization information
	processinfo	Process information
	procstats	Process statistics
	scheduling_policy	Thread scheduling
Recovery, Updates & Imaging (Chapter 6)	gsiservice	10.0: Generic System Image (GSI) service
	dynamic_system	11.0: Dynamic System Update (DSU)
	update_engine	ChromeOS A/B updater
	system_update	Retrieve/set system update info
	recovery	Recovery
	webviewupdate	WebView component independent update
Packages (II/2)	installd	Install/remove packages
	package[_native]	The Package Manager
	overlay	Runtime Resource Overlay (RRO)
	idmap	Resource ID/overlay package mapper
	otadexopt	Perform DEX→ART conversion after OTA
	rollback	Package rollback
	platform_compat[_native]	11.0: SDKVersion compatibility settings
	usagestats	Package usage statistics
Power Mgmt (Chapter 13)	power	Wake locks, power mgmt
	batterystats	Battery statistics
	batteryproperties	Battery Health
	battery	Battery services
	suspend_control	Device suspend state
	deviceidle	Doze
	thermalservice	Thermal management (prevents overheating)
	hardware_properties	Temperature, CPU usage and fan speeds
Storage (Chapter 5)	mount	`StorageManager`
	diskstats	Disk usage statistics
	storaged[_pri]	Storage Management Daemon
	storagestats	Storage statistics
	devicestoragemonitor	Low disk space notifications
	apexservice	Android Pony EXpress daemon
	vold	The Volume Daemon
Security - Key storage (III)	a.security.keystore	System keystore, Java API compatible
	gatekeeper	Security token issuance authority
Security - Authentication (III)	auth	11.0: Authentication service
	biometric	Biometric authentication
	face	"Face ID" Authentication
	fingerprint	Fingerprint authentication
Security - Authorization (III)	appops	Application operation permissions
	permission	Dalvik permission enforcement
	permissionmgr	11.0: Perm. grant, revocation, white-listing, etc.
	role	Package roles
	uri_grants	Manages package permissions to URIs

Table 9/5-1 (cont.): Android Services, categorized by functionality

Category	Service Name	Handles
Security - Miscellaneous (III)	sec_key_att_app_id_provider	Provides info about apps with a given UID
	lock_settings	Lock screen settings
	secure_element	Owned by `com.android.se`
	trust	Certificate Trust management
	file_integrity	11.0: File Integrity
	android.security.identity	11.0: Identity Management
	entropy	Mixes `/dev/random` entropy
Telephony (II/12)	phone	Phone functions
	isms	SMS messaging
	iphonesubinfo	Phone related subscriber information
	simphonebook	On-SIM phone book and contact list
	isub	Subscriber information
	telecom	Telephony manager services
	imms	MMS messaging
	emergency_affordance	11.0: emergency call functions
	telephony.registry	Telephony registry and notifications
	[r]cs	Rich Communicate Services (Messaging).
	ions	Opportunistic networking service
	carrier_config	Carrier settings configuration
	euicc_card_controller	eSIM services
	econtroller	Downloadable subscription metadata, etc
	sip	Session Initiation Protocol (VoIP) support
UI (II)	dreams	a.service.dreams.IDreamManager
	statusbar	Statusbar/widget interface
	uimode	Night mode, Car mode, etc
	wallpaper	Wallpaper setting/scaling etc
	voiceinteraction	Voice interaction (Hey Google, etc)
	accessibility	Accessibility services
	vrmanager	11.0:
OEM (III)	oem_lock	Interface with OEM locking
	persistent_data_block	Interface to persistent data partition

Note, that even with this many services, the table may be incomplete as vendors and ODMs often add additional services into the main service namespace. Further, the categorization above is far from perfect (as is evident by the "miscellaneous" category). It does, however, enable a divide-and-conquer approach: The work's remaining chapters try to follow along those lines. Each category and its services are detailed in a chapter or section (with exceptions made for media, wherein audio and graphics are treated separately, and security).

5.1. LocalServices

As previously noted, services in `system_server` usually use `publishBinderpublish()` to expose themselves to clients system-wide, but another option is to use `publishLocalService()`. This adds the service class to the `LocalServices` class, which holds the class references internally. The `LocalServices` are visible only within `system_server`, and thus any service calls are carried out through method calls in the same process, by obtaining the object reference from the `LocalServices` class, and then invoking the method.

Table 9/5-2 lists the `LocalServices` registrations in Android 11. Some of these are merely subsets of the Binder services discussed in Table 9/5-1, while others expose functionality deemed private:

Table 9/5-2: The `LocalServices` internal to `system_server`

Service	Purpose
`com.android.server.AlarmManagerInternal`	As `alarm`
`com.android.server.usage.AppStandbyInternal`	Track application idle state
`android.attention.AttentionManagerInternal`	Tracks when user attention is on screen
`android.view.autofill.AutofillManagerInternal`	As `autofill`
`android.os.BatteryManagerInternal`	As `battery`
`android.internal.os.CachedDeviceState.Readonly`	Caches device state changes
`c.a.s.camera.CameraServiceProxy`	as `media.camera.proxy`
`c.a.s.display.ColorDisplayServiceInternal`	As `color_display`
`c.a.s.contentcapture.ContentCaptureManagerInternal`	As `content_capture`
`com.android.server.pm.CrossProfileAppsInternal`	As `crossprofileapps`
`com.android.server.DeviceIdleInternal`	As `deviceidle`
`DeviceStorageMonitorInternal`	As `devicestoragemonitor`
`android.server.display.DisplayManagerInternal`	As `display`
`c.a.s.display.color.DisplayTransformManager`	Display color transformations
`com.android.server.dreams.DreamManagerInternal`	As `dreams`
`com.android.server.job.JobSchedulerInternal`	As `jobscheduler`
`com.android.server.lights.LightsManager`	As lights
`com.android.server.LooperStats`	Looper statistics (as `looper_stats`)
`a.s.notification.NotificationManagerInternal`	as `notification`
`OverlayManagerService`	Runtime Resource Overlay (as `overlay`)
`com.android.server.people.PeopleServiceInternal`	Manage people and conversations for apps
`com.android.server.PinnerService`	Pin important files in memory
`com.android.server.power.PowerManagerInternal`	`power` local interfacce
`com.android.server.timezone.RulesManagerService`	Time zone rules
`com.android.server.soundtrigger.SoundTriggerInternal`	As `sound_trigger`
`TwilightManager`	Twilight (evening) detection (timezone)
`com.android.server.UiModeManagerInternal`	As `uimode`
`com.android.server.usage.UsageStatsManagerInternal`	As `usagestats`
`c.a.s.voiceinternaction.VoiceInteractionManagerInternal`	as `voiceinteraction`
`com.android.server.vr.VrManagerInternal`	Virtual Reality Manager