In a Bind?

Android’s Binder – in depth

Jonathan Levin

http://NewAndroidBook.com/
http://Technologeeks.com/

(c) 2015 Jonathan Levin, NewAndroidBook.com, licensed to Technologeeks.com
IPC, Illustrated
Some definitions

• IPC: Inter Process Communication
  – Allowing two separate processes to interact
    • Otherwise, process virtual memory is entirely isolated
  – Many mechanisms exist:
    • Message passing
    • Sockets
    • Shared mem

• RPC: Remote Procedure Call
  – Hides IPC inside a function call.
Some definitions

• Proxy: The client end of the RPC
  – Serializes arguments, sends message
  – Gets reply, deserializes return code/exception

• Stub: The server end of the RPC
  – Deserializes arguments
  – Invokes real function (hence, itself, a stub)
  – Serializes return code/exception
Some definitions

• **Endpoint Mapper:**
  - Allows unrelated clients and servers to talk
    • A priori - Known to all, loved by all
    • Servers register with it
    • Clients lookup with it (Context.getSystemService)
The Service Manager

- Android’s Endpoint mapper
- Single threaded process, started by Init
- Designated as system critical
  - If it fails – soft system restart (zygote, all services)
The Service Manager

A) Client calls getService API (usually, through Context.getSystemService(...), which wraps android.os.ServiceManager.getService())

B) A SVC_MGR_GET_SERVICE transaction is sent over Binder to the context manager

C) servicemanager traverses the svclist, looking for a match on name, and returns the reference (stored in “ptr”) to the caller

D) Newly allocated entry is added to the svclist linked list

3) servicemanager receives the message. Allocates a new svcinfo structure, and populates it with the name, and service reference obtained from message.

4) Linked list pointer to next entry

(c) 2015 Jonathan Levin, NewAndroidBook.com, licensed to Technoloceeks.com
The Service Manager

1) User code calls Context.getSystemService()

2) Returned object, type cast to a "Manager" contains a private binder handle to service

3) Service calls are proxied by the Manager class, which serializes RPC over binder, using AIDL

0) Service binder handle is registered a priori by system_server (or another 3rd party service process)
Binder

• Binder provides the core of RPC in Android
  – Provides conduit to pass intents and other objects
  – Supports remote method invocation (AIDL)

• UNIX sockets still used, but infrequently
  – Zygote and some native daemons (/dev/socket/*)
  – InputDispatcher to Applications (socketpairs)
Binder

• UNIX sockets have serious shortcomings:
  – Anonymous sockets can only be shared by progeny
    • Inherited as file descriptors
  – Named sockets require a filesystem “presence”
    • Filesystem ensures system-wide uniqueness and visibility
    • This requires a writable location – not many of those..
    • Also vulnerable to race conditions (open to MiM attacks)
Binder

• Binder provides an alternative to sockets
  – Supports most socket functionality
    • Credentials
    • Passing descriptors
      – Can also pass shared memory (using ashmem descriptors)

  – Extends capabilities to a full IPC/RPC model
    • Allows dynamic registration and location of services
    • Provides “Death Notifications”
Binder Nomenclature

• A STRONG reference is a handle to an object
  – Object remains alive so long as >=1 STRONG exist
  – A handle can only be used (called) if it is strongly held

• A WEAK reference is just a “link” to an object
  – Essentially, a declaration of “interest” in the object
  – Object may actually die, in which case weak ref is voided
  – Reference must be promoted to a STRONG ref to be used

• WEAK references enable Death Notifications
Death Notifications

• Binder will register your interest in an object
  – Effectively, providing you with a weak reference to it

• If object dies (host process exits or killed):
  – Remote Binder sends you a notification (obituary)
  – Your local Binder calls your callback
Binder

• Android goes to great lengths to abstract Binder
  – Java Layer: AIDL
  – Native Layer: libBinder.cpp

• Actual mechanism is implemented by kernel module
  – For the longest time, in drivers/staging, now in mainline.
• Developer starts by writing method implementations
• Methods and objects are exported into an .aidl file

```
package com.NewAndroidBook.example;

interface ISample {
    /* 1 */ void someFunc (int someArg); // no return value, integer arg
    /* 2 */ boolean anotherFunc(String someArg); // boolean return value, string arg
}
```

• AIDL is a bit like a C header file (.h) – just prototypes
• The SDK “aidl” tool auto generates client/server code:
  – “Proxy” for the client: serializes arguments, invokes method
  – “Stub” for the server: deserializes, calls, serializes return value

(c) 2015 Jonathan Levin, NewAndroidBook.com, licensed to Technologeeks.com
• The framework abstracts both classes with an “IBinder”
  – The “BinderProxy” serves as client, exports “transact()”
  – The “Binder” provides an “onTransact()” callback

• “transact()” magically invokes remote “onTransact()”
• Transaction can carry a serializeable “Parcel” object
• The Framework uses JNI to communicate with library
• The “JavaBBinder” object bridges upcalls back to VM
libBinder provides matching native level objects:
- IBinder ("interface"), Bbinder, BpBinder, and Parcel
Java (Developer)

AIDL

Java Proxy Class (generated by AIDL)

android.os.IBinder:transact()

Method implementation (provided by developer)

Java Stub Class (generated by AIDL)

Generated from class.aidl

Framework

android.os.BinderProxy:transactNative()

android.os.Parcel

android.os.Binder:execTransact()

BBinder:transact()

IPCThreadState:executeCommand()

Native

android_os_BinderProxy_transact()

IBinder:transact()

BpBinder:transact()

IPCThreadState::self()->transact()

ioctl(2)

void *Data

IPCThreadState:executeCommand()

IPCThreadState.cpp

Android

android_util_Binder.cpp

libBinder.so (Binder.cpp)

ProcessState/IPCThreadState abstract kernel interface

(c) 2015 Jonathan Levin, NewAndroidBook.com, licensed to Technologeeks.com
Method implementation (provided by developer)

Java Stub Class
(generated by AIDL)

Java Proxy Class
(generated by AIDL)

android.os.IBinder:transact()

(android.os.Binder:transact())

(android.os.Binder:execTransact())

BBinder::onTransact()

IPCThreadState:executeCommand()

BBinder::onTransact()

IPCThreadState::self()->transact()

IOCTLI(2)

/dev/binder Kernel Module

(c) 2015 Jonathan Levin, NewAndroidBook.com, licensed to Technologeeks.com
Layers::AIDL

- Demo: AIDL code generation
Layers::Framework

- Android.os.Binder is actually quite documented
- Defines the “default transactions”*

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Default Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>PING_TRANSACTION</td>
<td>_PNG</td>
<td>Null transaction ensuring service object is alive. (q.v. android.os.IBinder.pingBinder()</td>
</tr>
<tr>
<td>DUMP_TRANSACTION</td>
<td>_DMP</td>
<td>Requests full dump of service state. Used by dumpsys</td>
</tr>
<tr>
<td>INTERFACE_TRANSACTION</td>
<td>_NTF</td>
<td>Requests interface of service object behind handle. Expects UTF-16 interface name as reply</td>
</tr>
<tr>
<td>SYSPROPS_TRANSACTION</td>
<td>_SPR</td>
<td>Used by native code only: calls libutils's report_sysprop_change(), which invokes any registered callbacks</td>
</tr>
</tbody>
</table>

- Meant to be overridden
  - AIDL code does that automatically for you

* - And a couple of not-so-funny joke transactions as well (TWEET, LIKE..)
Layers::LibBinder

- LibBinder isn’t documented at all..
- Object structure essentially mirrors Java’s
- Excessively heavy use of templates, macros
  - Not trivial to follow class hierarchy/flow at all..
Base class for strong and weak references
Also provides wp<> and sp<>
Layers::LibBinder

- ProcessState/IPCThreadState further abstract:
  - Actual kernel interface entirely hidden
  - Thread Pool Management
Layers::LibBinder

• Apps (read: Zygote) automatically start pool
  – frameworks/base/cmds/app_process/app_main.cpp

• Native services work similarly
  – Examples: healthd, bootanimation, InputFlinger(!)
Layers::Kernel

• At the kernel level, Binder is a character device
  – Created as “misc” device, world writable

• All communication done via ioctl(2) calls
  – No read/write needed
  – Clients open, configure with ioctl(2) then either:
    • Enter a polling loop (IPCThreadState::setupPolling)
    • Block until message/timeout (IPCThread::joinThreadPool)
int BinderFD = open("/dev/binder")

Ensures Binder kernel API version matches libBinder’s

Set Max Thread pool size to libBinder default (15)

Map Transaction Memory (1MB – 8K)

Main thread Enters looper, secondaries register with it

Write command to driver, optionally block for read

Secondary threads may exit on time out if not needed

Notify Binder of thread termination
Layers::Kernel

• `ioctl(2)` buffer points to read/write buffers
  – Write buffers provide BC_ commands to Binder
    • These may contain transactions, for Binder to execute

  – Read buffers provide BR_ replies from Binder
    • These may contain transactions for clients to execute
    • May also contain “death notifications”

• Buffers are optional (size specified may be 0)
Layers::Kernel

- write_size: Size of buffer (specified by client)
- write_consumed: Size of buffer (used up by driver)
- write_buffer
- read_size: Size of buffer (specified by client)
- read_consumed: Size of buffer (used up by driver)
- read_buffer

BC_cmd
- Command Data
- ..., BR_Reply
- Reply Data
- ..., ...

(c) 2015 Jonathan Levin, NewAndroidBook.com, licensed to Technologeeks.com
<table>
<thead>
<tr>
<th>BC_code</th>
<th>Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCREFS</td>
<td></td>
<td>Increase reference count of argument</td>
</tr>
<tr>
<td>DECREFS</td>
<td>Handle</td>
<td>Decrease reference count of argument</td>
</tr>
<tr>
<td>ACQUIRE</td>
<td></td>
<td>Acquire a Binder reference</td>
</tr>
<tr>
<td>RELEASE</td>
<td></td>
<td>Release a Binder reference</td>
</tr>
<tr>
<td>INCREFS_DONE</td>
<td>Refs, obj</td>
<td>Informs Binder reference has been increased</td>
</tr>
<tr>
<td>ACQUIRE_DONE</td>
<td></td>
<td>Informs Binder reference has been acquired</td>
</tr>
<tr>
<td>ATTEMPT_ACQUIRE</td>
<td></td>
<td>Attempts to acquire a reference (not supported)</td>
</tr>
<tr>
<td>ACQUIRE_RESULT</td>
<td></td>
<td>Informs Binder as to success of attempted acquire (not supported)</td>
</tr>
<tr>
<td>FREE_BUFFER</td>
<td>void *</td>
<td>Informs Binder the buffer provided may be safely freed</td>
</tr>
<tr>
<td>TRANSACTION</td>
<td></td>
<td>Contains additional Binder transaction data</td>
</tr>
<tr>
<td>REPLY</td>
<td></td>
<td>Contains additional Binder transaction data</td>
</tr>
<tr>
<td>REGISTER_LOOPER</td>
<td></td>
<td>Called by secondary threads entering the Binder thread pool</td>
</tr>
<tr>
<td>ENTER_LOOPER</td>
<td></td>
<td>Called by the main thread when entering the Binder thread pool</td>
</tr>
<tr>
<td>EXIT_LOOPER</td>
<td></td>
<td>Called by any threads exiting the Binder thread pool (usually as result of timeout)</td>
</tr>
<tr>
<td>REQUEST_DEATH_NOTIFICATION</td>
<td>Handle, Proxy</td>
<td>Informs Binder client is interested in receiving notifications when remote process terminates, for whatever reason.</td>
</tr>
<tr>
<td>CLEAR_DEATH_NOTIFICATION</td>
<td></td>
<td>Informs Binder client is no longer interested in death notification for remote process</td>
</tr>
<tr>
<td>DEAD_BINDER_DONE</td>
<td>Proxy</td>
<td>Reply to Binder death notification, informing Binder the reference death notification has been processed.</td>
</tr>
</tbody>
</table>
Driver Protocol::Replies

<table>
<thead>
<tr>
<th>BR_code</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR</td>
<td>Informs client of some Binder error</td>
</tr>
<tr>
<td>OK</td>
<td>Informs client everything is ok</td>
</tr>
<tr>
<td>NOOP</td>
<td>No operation</td>
</tr>
<tr>
<td>INCREFS</td>
<td>Increase reference count of argument</td>
</tr>
<tr>
<td>DECREFS</td>
<td>Decrease reference count of argument</td>
</tr>
<tr>
<td>ACQUIRE</td>
<td>Acquire a Binder reference</td>
</tr>
<tr>
<td>RELEASE</td>
<td>Release a Binder reference</td>
</tr>
<tr>
<td>ATTEMPT_ACQUIRE</td>
<td>Attempts to acquire a reference (not supported)</td>
</tr>
<tr>
<td>ACQUIRE_RESULT</td>
<td>Informs Binder as to success of attempted acquire (not supported)</td>
</tr>
<tr>
<td>TRANSACTION</td>
<td>Incoming transaction requested of the client</td>
</tr>
<tr>
<td>REPLY</td>
<td>Result of previous transaction requested by the client</td>
</tr>
<tr>
<td>TRANSACTION_COMPLETE</td>
<td></td>
</tr>
<tr>
<td>SPAWN_LOOPER</td>
<td>Informing client a thread is required</td>
</tr>
<tr>
<td>FINISHED</td>
<td></td>
</tr>
<tr>
<td>DEAD_BINDER</td>
<td></td>
</tr>
<tr>
<td>DEAD_REPLY</td>
<td></td>
</tr>
<tr>
<td>FAILED_REPLY</td>
<td></td>
</tr>
<tr>
<td>CLEAR_DEATH_NOTIFICATION_DONE</td>
<td></td>
</tr>
</tbody>
</table>
# Binder Transaction Data

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>32-bit handle or pointer</td>
</tr>
<tr>
<td>cookie</td>
<td>Used to detect mismatched handles</td>
</tr>
<tr>
<td>code</td>
<td>Transaction code. One of the built-in codes, or application defined</td>
</tr>
<tr>
<td>flags</td>
<td>TF_flags, indicating ONE WAY, ACCEPT FDS, or STATUS CODE (ROOT_OBJECT unused)</td>
</tr>
<tr>
<td>sender_pid</td>
<td>Process identifier of sender</td>
</tr>
<tr>
<td>sender_uid</td>
<td>UID of message sender</td>
</tr>
<tr>
<td>data_size</td>
<td>If non zero, indicates data is a pointer to buffer of this number of bytes</td>
</tr>
<tr>
<td>offsets_size</td>
<td>If non zero, indicates data provides offsets into this message</td>
</tr>
<tr>
<td>data</td>
<td>Pointer to a buffer of data_size bytes, or offsets into message</td>
</tr>
</tbody>
</table>
ProcessState::self()

Creates the singleton (on first invocation) or returns it.

ProcessState::startThreadPool()

ProcessState::spawnPooledThread()

new PoolThread(isMain)

new Thread()

Creates the thread (using androidCreateRawThreadEtc) and enters the threadLoop

PoolThread::threadLoop()

Inform Binder (BC\_\_LOOPER)

Main reports ENTER, seconaries report REGISTER

processPendingDerefs()

getAndExecuteCommand()

Inform Binder (BC\_EXIT\_LOOPER)

IPCThreadState::threadDestructor

BR\_\_DECREFS/B\_\_RELEASE

Sets pending derefs

Spawns secondary

Upcalls incoming transactions

BR\_\_SPAWN\_\_LOOPER

BR\_\_TRANSACTION

BR\_\_DEAD\_\_BINDER

BR\_\_

BpBinder::sendObituary()

BBinder::transact()

Handles Death Notifications
Tracing/Debugging

Experiment: Using the bindump tool to view open Binder handles

The bindump tool, which you can find on the Book’s companion website is nothing more than a simple derivative of the service command, which obtains a handle to the system service of choice (as does service check), and then inspects its own entry in the /sys/kernel/debug/bind/binder/proc directory. 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. Because all the Binder debug data is world readable, you can run this tool on unrooted devices as well.

Output 6-3: Revealing Binder endpoints using the bindump utility

```
# Inquire about wallpaper service
shell@htc_m8w1l: $ /data/local/tmp/bindump wallpaper
Service: wallpaper node refs 2034
User: PID 1377 com.htc.launcher
User: PID 1194 com.android.systemui
User: PID 368 /system/bin/service_manager

# Who owns the batterypropreg service?
shell@htc_m8w1l: $ /data/local/tmp/bindump owner batterypropreg
Service: batterypropreg node refs 105765
Owner: PID 8153 /sbin/healthd
```

Another free tool to monitor Binder connections is Opsys's Binder Explorer. This tool works as an App, or along with an HTML GUI, to show a graphical view of connections in real time.

The book’s companion website also provides jtrace, with is a special version of strace(1), the Linux system call tracing tool, with augmented functionality that includes parsing of Binder messages (i.e. deciphering ioctl(2) codes and payloads).
That’s (NOT) All, Folks!
@Technologeeks Training

- “Android Internals & Reverse Engineering” training discusses all this, and more
  - Native level debugging and tracing
  - Binder internals
  - Native services
  - Frameworks
  - DEX, OAT structure and reversing
  - Graphics, Media and Audio

- Based on “Android Internals” Volume I and (Jan ’16, finally!) Volume II

- [http://Technologeeks.com/AIRE](http://Technologeeks.com/AIRE)
  - Next training: February 8th, 2016, NYC!

- Follow @Technologeeks for updates, training, and more!
Some References

Great discussion:


Old, but nice:

- https://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf

My book:

- Android Internals, Volume II, Ch. 11 (http://NewAndroidBook.com/)