In a Bind?

Android's Binder – in depth

Jonathan Levin

http://NewAndroidBook.com/

http://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)

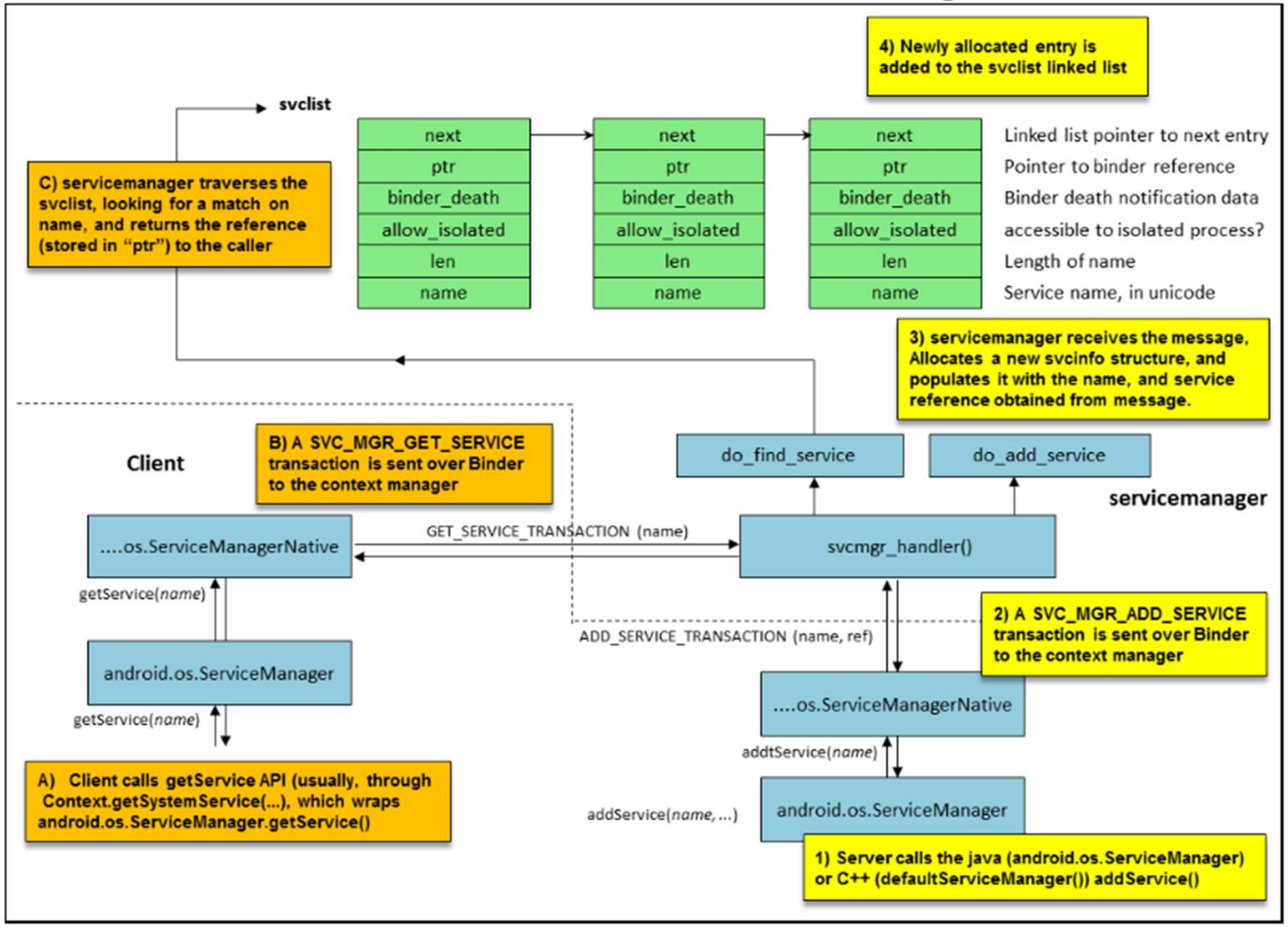
The Service Manager

- Android's Endpoint mapper

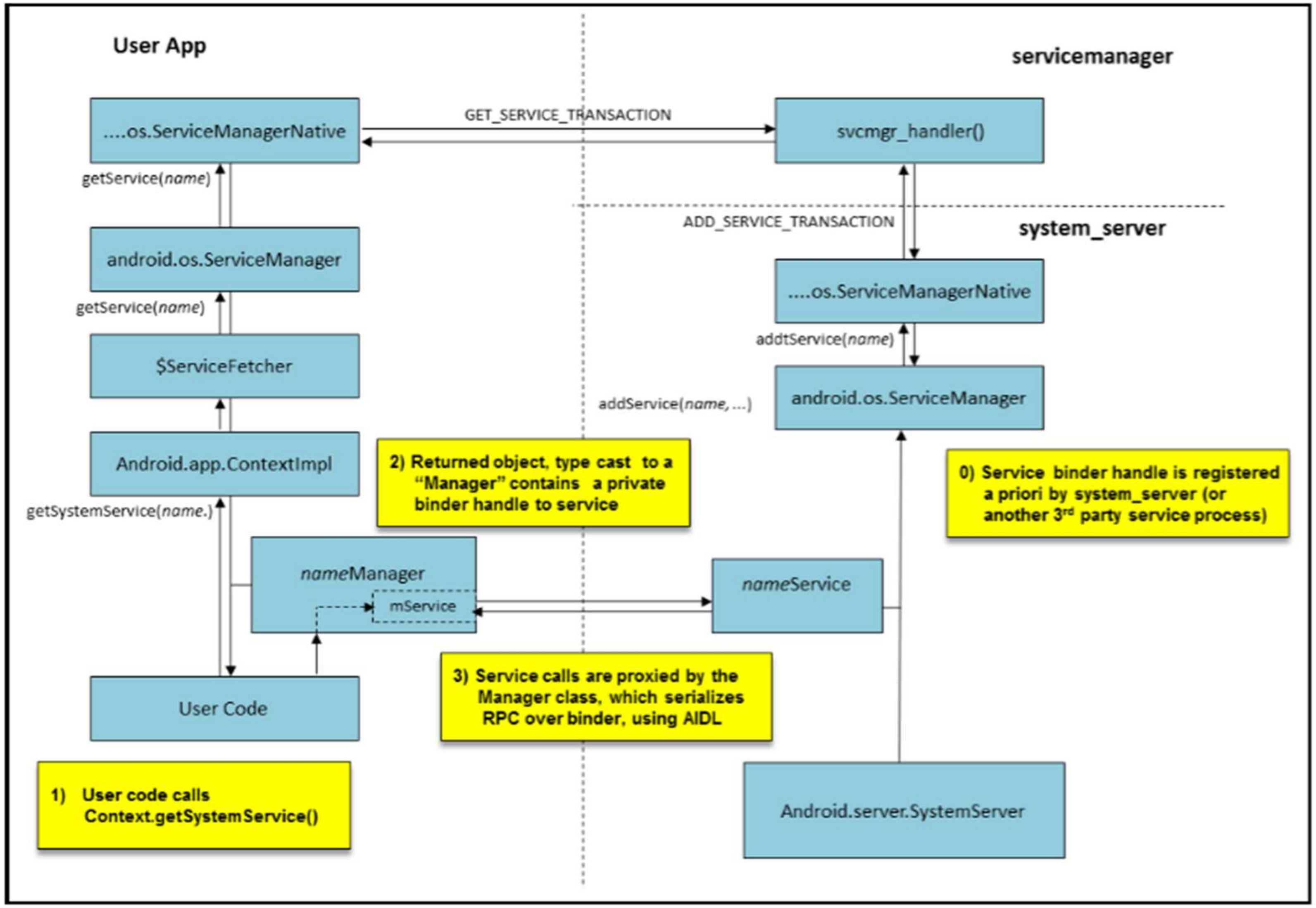
- Single threaded process, started by Init

- Designated as system critical
  - If it fails – soft system restart (zygote, all services)

The Service Manager


4) Newly allocated entry is added to the svclist linked list
svclist
C) servicemanager traverses the svclist, looking for a match on name, and returns the reference (stored in "ptr") to the caller
next
ptr
binder_death
allow_isolated
len
name
Linked list pointer to next entry
Pointer to binder reference
Binder death notification data
accessible to isolated process?
Length of name
Service name, in unicode
3) servicemanager receives the message, Allocates a new svcinfo structure, and populates it with the name, and service reference obtained from message.
Client
B) A SVC_MGR_GET_SERVICE transaction is sent over Binder to the context manager
do_find_service
do_add_service
servicemanager
....os.ServiceManagerNative
GET_SERVICE_TRANSACTION (name)
svcmgr_handler()
getService(name)
2) A SVC_MGR_ADD_SERVICE transaction is sent over Binder to the context manager
ADD_SERVICE_TRANSACTION (name, ref)
android.os.ServiceManager
....os.ServiceManagerNative
getService(name)
addtService(name)
A) Client calls getService API (usually, through Context.getSystemService(...), which wraps android.os.ServiceManager.getService()
addService(name, ...)
android.os.ServiceManager
1) Server calls the java (android.os.ServiceManager) or C++ (defaultServiceManager()) addService()

The Service Manager



User App
servicemanager
....os.ServiceManagerNative
GET_SERVICE_TRANSACTION
svcmgr_handler()
getService(*name*)
ADD_SERVICE_TRANSACTION
system_server
android.os.ServiceManager
getService(*name*)
....os.ServiceManagerNative
$ServiceFetcher
addtService(*name*)
addService(*name*, ...)
android.os.ServiceManager
Android.app.ContextImpl
2) Returned object, type cast to a "Manager" contains a private binder handle to service
0) Service binder handle is registered a priori by system_server (or another 3rd party service process)
getSystemService(*name*.)
*name*Manager
mService
*name*Service
User Code
3) Service calls are proxied by the Manager class, which serializes RPC over binder, using AIDL
1) User code calls Context.getSystemService()
Android.server.SystemServer

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
  – Framework Layer: Android.os.Binder (+ android_util_Binder)
  – Native Layer: libBinder.cpp

- Actual mechanism is implemented by kernel module
  – For the longest time, in drivers/staging, now in mainline.

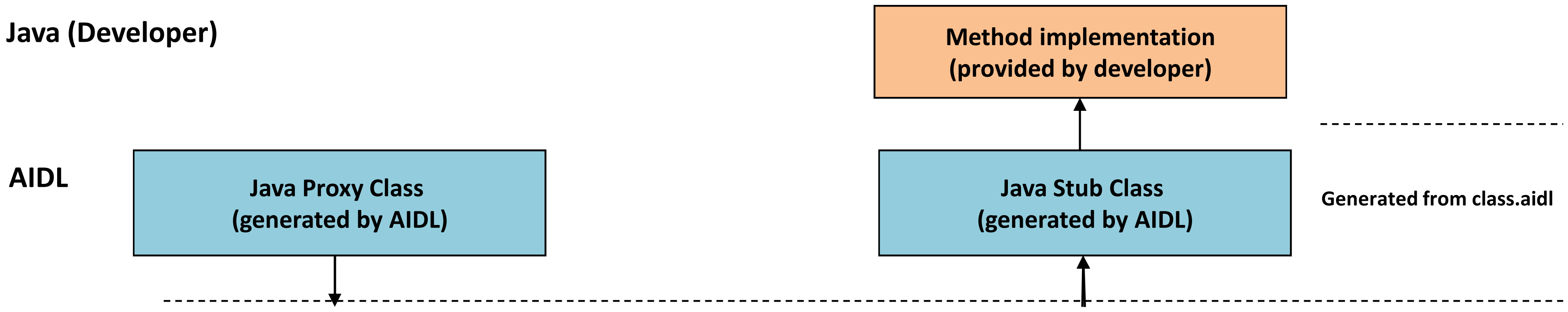
Java (Developer)
Method implementation
(provided by developer)

- Developer starts by writing method implementations
- Methods and objects are exported into an .aidl file

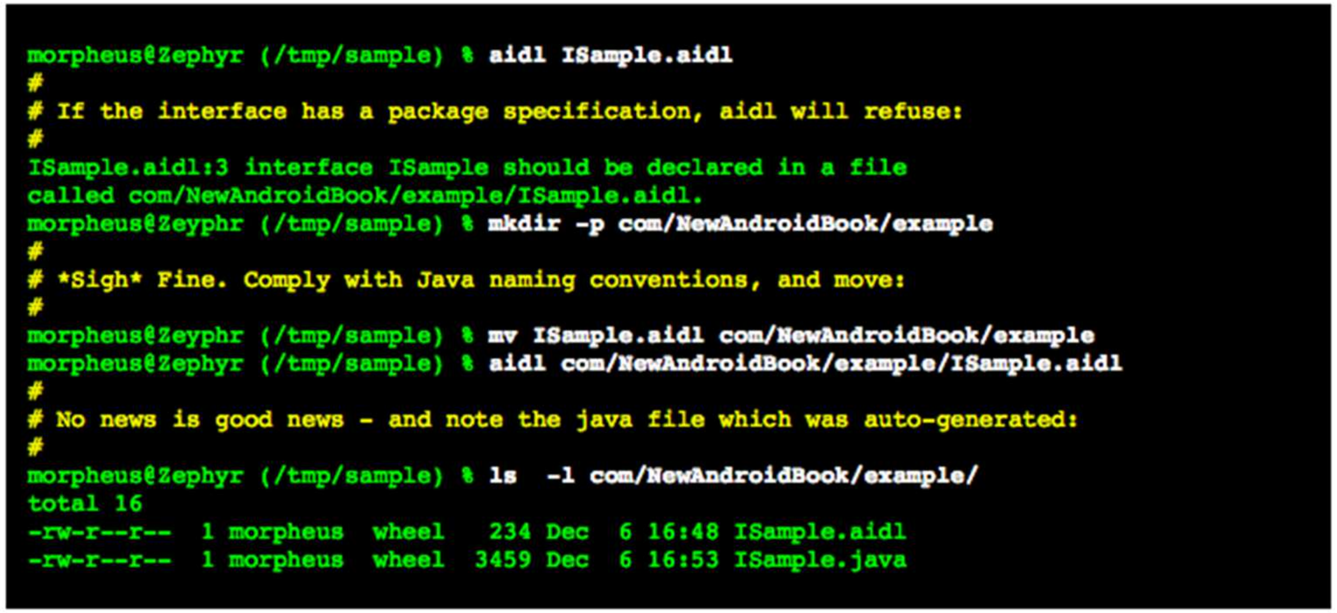
```
package com.NewAndroidBook.example;

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

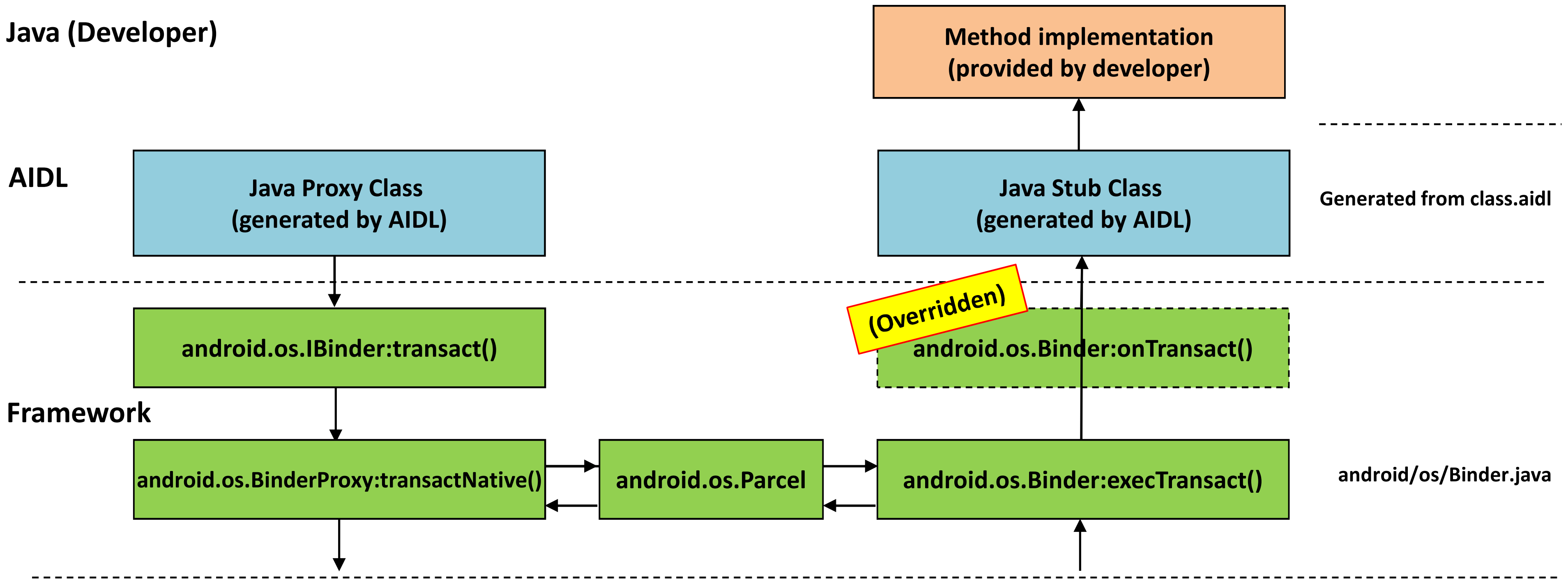
- AIDL is a bit like a C header file (.h) – just prototypes


Java (Developer)
Method implementation (provided by developer)
AIDL
Java Proxy Class (generated by AIDL)
Java Stub Class (generated by AIDL)
Generated from class.aidl

- 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

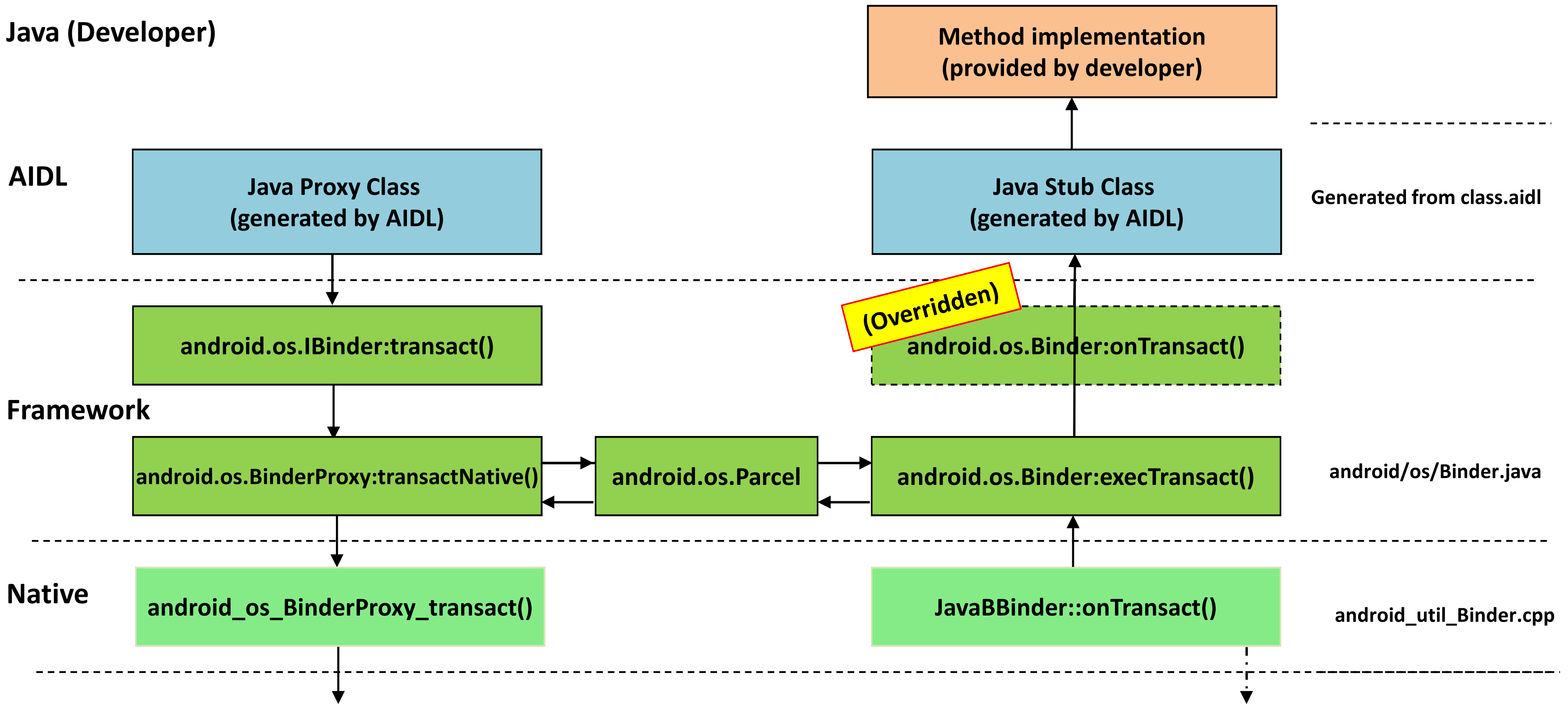


```
morpheus@Zephyr (/tmp/sample) % aidl ISample.aidl
#
# If the interface has a package specification, aidl will refuse:
#
ISample.aidl:3 interface ISample should be declared in a file
called com/NewAndroidBook/example/ISample.aidl.
morpheus@Zeyphr (/tmp/sample) % mkdir -p com/NewAndroidBook/example
#
# *Sigh* Fine. Comply with Java naming conventions, and move:
#
morpheus@Zeyphr (/tmp/sample) % mv ISample.aidl com/NewAndroidBook/example
morpheus@Zephyr (/tmp/sample) % aidl com/NewAndroidBook/example/ISample.aidl
#
# No news is good news - and note the java file which was auto-generated:
#
morpheus@Zephyr (/tmp/sample) % ls  -l com/NewAndroidBook/example/
total 16
-rw-r--r--  1 morpheus  wheel   234 Dec  6 16:48 ISample.aidl
-rw-r--r--  1 morpheus  wheel  3459 Dec  6 16:53 ISample.java
```


Java (Developer)
Method implementation (provided by developer)
AIDL
Java Proxy Class (generated by AIDL)
Java Stub Class (generated by AIDL)
Generated from class.aidl
(Overridden)
android.os.IBinder:transact()
android.os.Binder:onTransact()
Framework
android.os.BinderProxy:transactNative()
android.os.Parcel
android.os.Binder:execTransact()
android/os/Binder.java

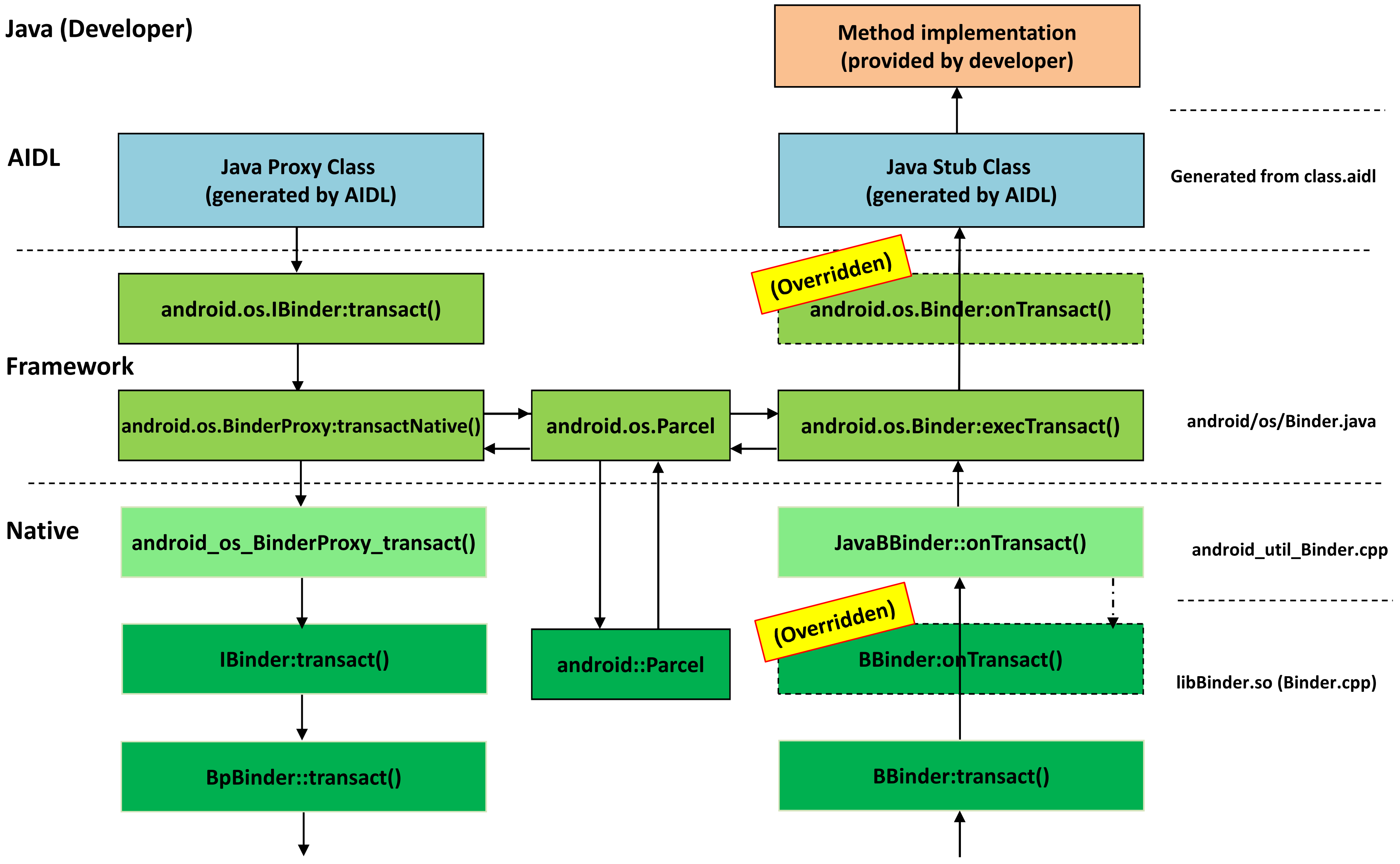
- The framework abstracts both classes with an “IBinder”
  - The “BinderProxy” serves as client, exports “transact()”
  - The “Binder” provides an “onTransact()” callback

- “transact()” magically invokes remote “onTransact()”
- Transaction can carry a serializeable “Parcel” object

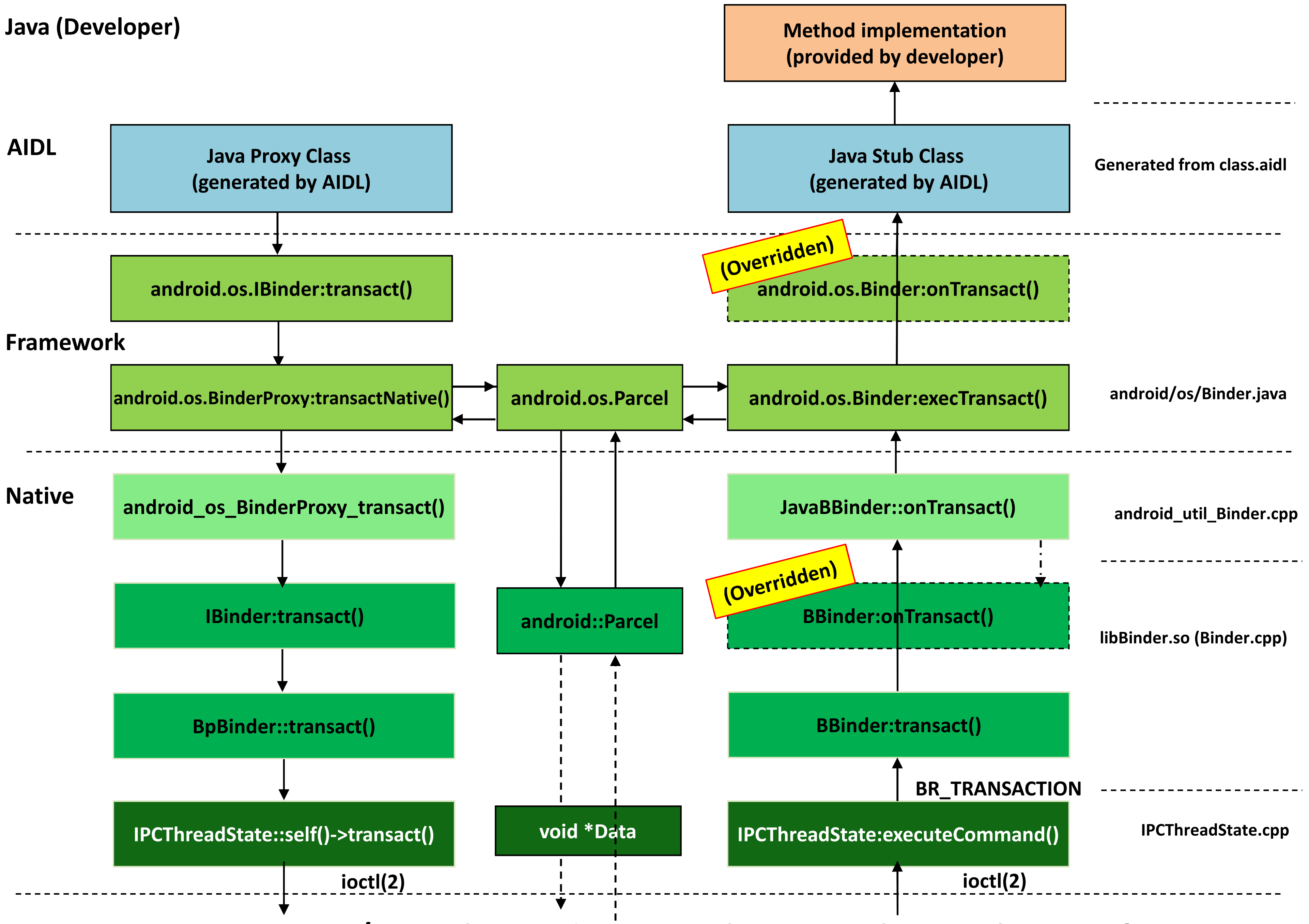


Java (Developer)
Method implementation (provided by developer)
AIDL
Java Proxy Class (generated by AIDL)
Java Stub Class (generated by AIDL)
Generated from class.aidl
(Overridden)
android.os.IBinder:transact()
android.os.Binder:onTransact()
Framework
android.os.BinderProxy:transactNative()
android.os.Parcel
android.os.Binder:execTransact()
android/os/Binder.java
Native
android_os_BinderProxy_transact()
JavaBBinder::onTransact()
android_util_Binder.cpp

- The Framework uses JNI to communicate with library
- The “JavaBBinder” object bridges upcalls back to VM

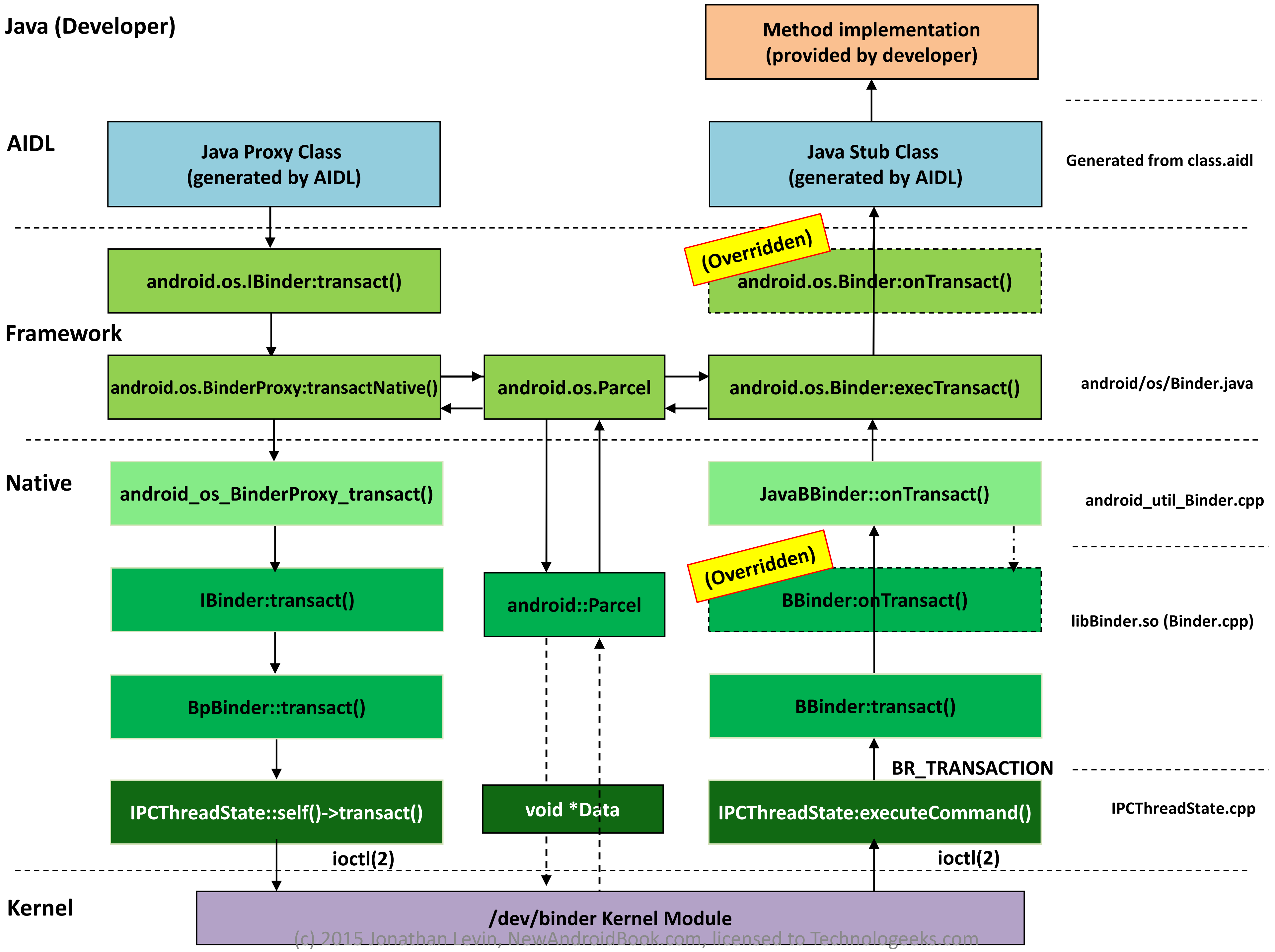

Java (Developer)
Method implementation (provided by developer)
AIDL
Java Proxy Class (generated by AIDL)
Java Stub Class (generated by AIDL)
Generated from class.aidl
(Overridden)
android.os.IBinder:transact()
android.os.Binder:onTransact()
Framework
android.os.BinderProxy:transactNative()
android.os.Parcel
android.os.Binder:execTransact()
android/os/Binder.java
Native
android_os_BinderProxy_transact()
JavaBBinder::onTransact()
android_util_Binder.cpp
(Overridden)
IBinder:transact()
android::Parcel
BBinder:onTransact()
libBinder.so (Binder.cpp)
BpBinder::transact()
BBinder:transact()

- libBinder provides matching native level objects:
  - Ibinder (“interface”), Bbinder, BpBinder, and Parcel



Java (Developer)
Method implementation (provided by developer)
AIDL
Java Proxy Class (generated by AIDL)
Java Stub Class (generated by AIDL)
Generated from class.aidl
(Overridden)
android.os.IBinder:transact()
android.os.Binder:onTransact()
Framework
android.os.BinderProxy:transactNative()
android.os.Parcel
android.os.Binder:execTransact()
android/os/Binder.java
Native
android_os_BinderProxy_transact()
JavaBBinder::onTransact()
android_util_Binder.cpp
IBinder:transact()
android::Parcel
(Overridden)
BBinder:onTransact()
libBinder.so (Binder.cpp)
BpBinder::transact()
BBinder:transact()
BR_TRANSACTION
IPCThreadState::self()->transact()
void *Data
IPCThreadState:executeCommand()
IPCThreadState.cpp
ioctl(2)
ioctl(2)

- ProcessState/IPCThreadState abstract kernel interface

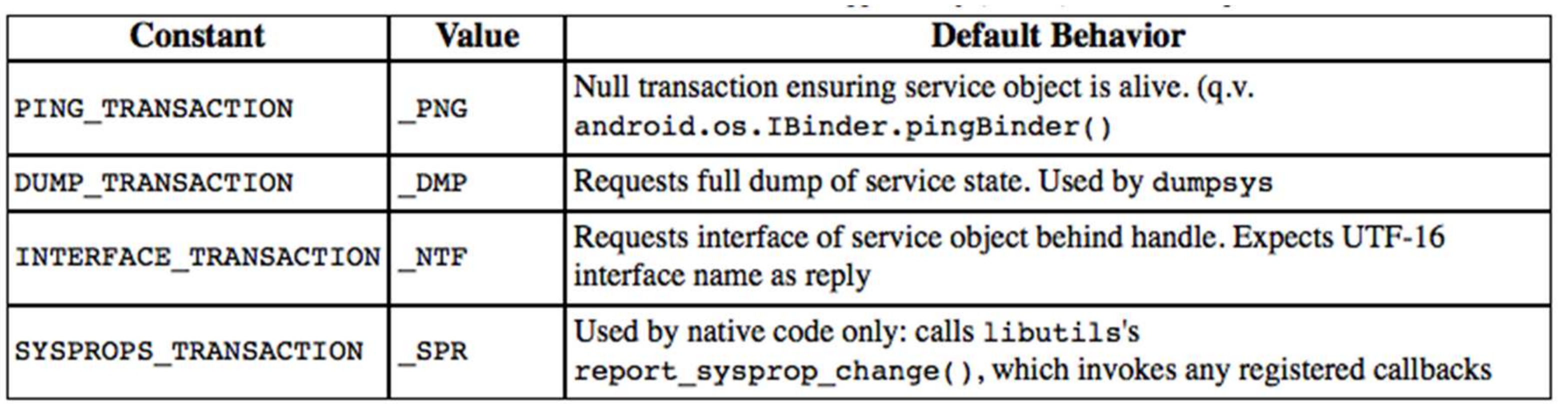

Java (Developer)
Method implementation (provided by developer)
AIDL
Java Proxy Class (generated by AIDL)
Java Stub Class (generated by AIDL)
Generated from class.aidl
(Overridden)
android.os.IBinder:transact()
android.os.Binder:onTransact()
Framework
android.os.BinderProxy:transactNative()
android.os.Parcel
android.os.Binder:execTransact()
android/os/Binder.java
Native
android_os_BinderProxy_transact()
JavaBBinder::onTransact()
android_util_Binder.cpp
(Overridden)
IBinder:transact()
android::Parcel
BBinder:onTransact()
libBinder.so (Binder.cpp)
BpBinder::transact()
BBinder:transact()
BR_TRANSACTION
IPCThreadState::self()->transact()
void *Data
IPCThreadState:executeCommand()
IPCThreadState.cpp
ioctl(2)
ioctl(2)
Kernel
/dev/binder Kernel Module

Layers::AIDL

- Demo: AIDL code generation

Layers::Framework

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



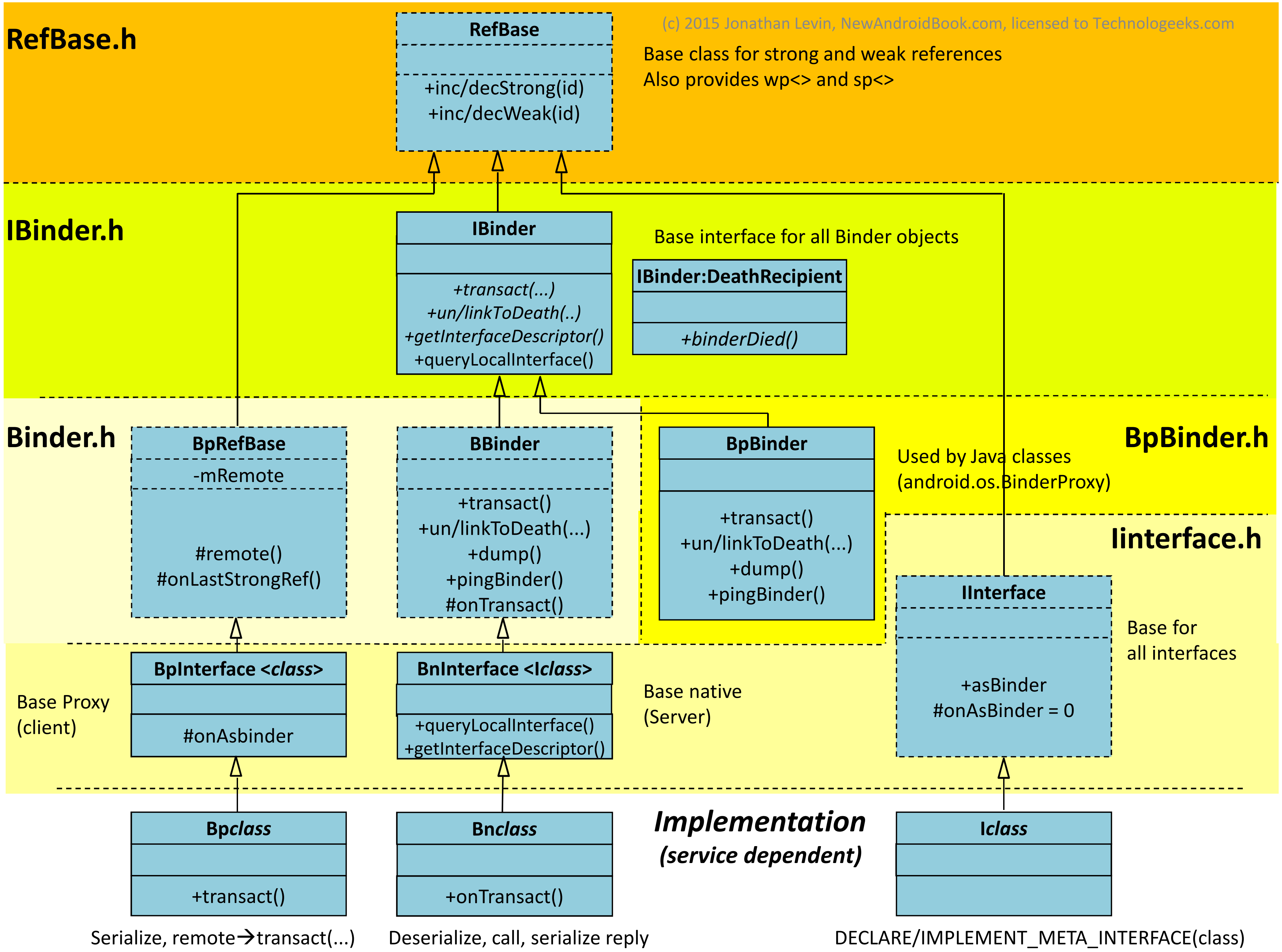
Constant	Value	Default Behavior
PING_TRANSACTION	_PNG	Null transaction ensuring service object is alive. (q.v. android.os.IBinder.pingBinder()
DUMP_TRANSACTION	_DMP	Requests full dump of service state. Used by dumpsys
INTERFACE_TRANSACTION	_NTF	Requests interface of service object behind handle. Expects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

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

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

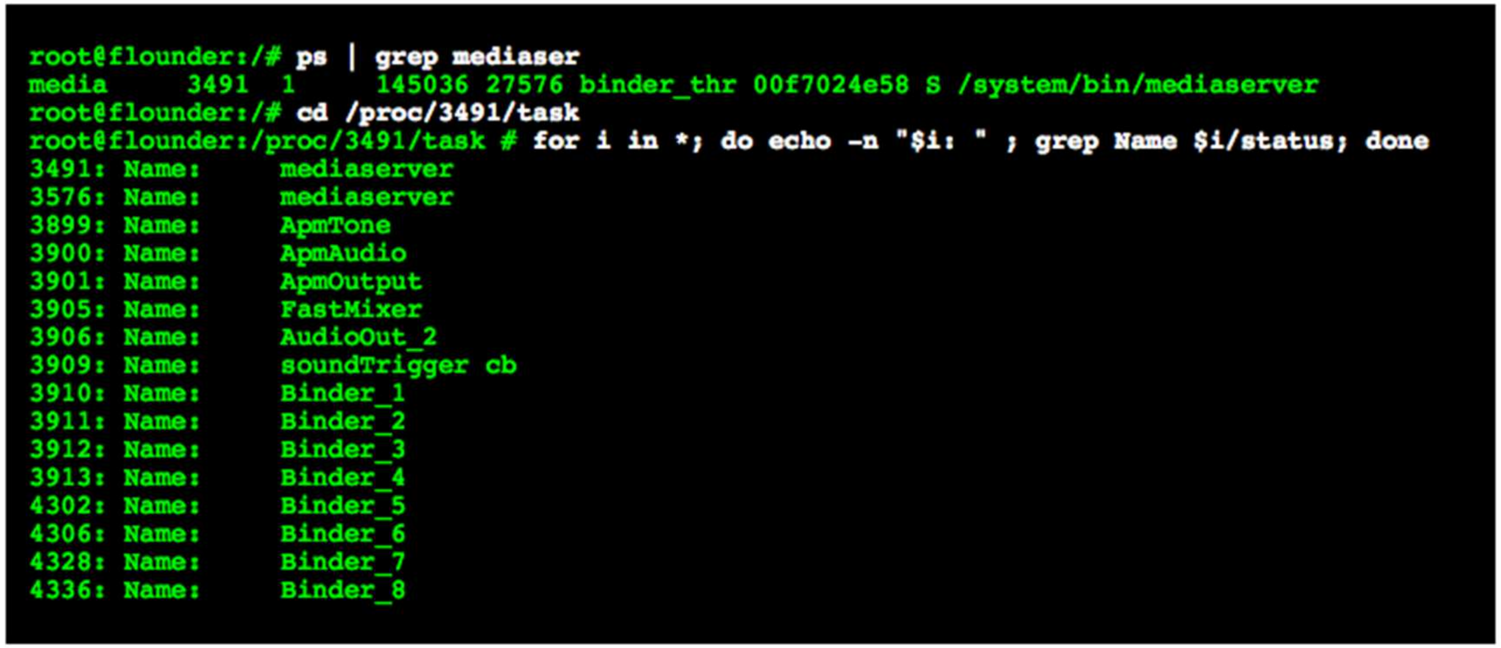
Layers::LibBinder

- 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..


RefBase.h
RefBase
+inc/decStrong(id)
+inc/decWeak(id)
(c) 2015 Jonathan Levin, NewAndroidBook.com, licensed to Technologeeks.com
Base class for strong and weak references
Also provides wp<> and sp<>
IBinder.h
IBinder
+transact(...)
+un/linkToDeath(..)
+getInterfaceDescriptor()
+queryLocalInterface()
Base interface for all Binder objects
IBinder:DeathRecipient
+binderDied()
Binder.h
BpRefBase
-mRemote
#remote()
#onLastStrongRef()
BBinder
+transact()
+un/linkToDeath(...)
+dump()
+pingBinder()
#onTransact()
BpBinder
+transact()
+un/linkToDeath(...)
+dump()
+pingBinder()
Used by Java classes
(android.os.BinderProxy)
BpBinder.h
Iinterface.h
IInterface
+asBinder
#onAsBinder = 0
Base for
all interfaces
Base Proxy
(client)
BpInterface <class>
#onAsbinder
BnInterface <Iclass>
+queryLocalInterface()
+getInterfaceDescriptor()
Base native
(Server)
Bpclass
+transact()
Bnclass
+onTransact()
Implementation
(service dependent)
Iclass
Serialize, remote→transact(...)
Deserialize, call, serialize reply
DECLARE/IMPLEMENT_META_INTERFACE(class)

Layers::LibBinder

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



```
root@flounder:/# ps | grep mediaser
media     3491  1     145036 27576 binder_thr 00f7024e58 S /system/bin/mediaserver
root@flounder:/# cd /proc/3491/task
root@flounder:/proc/3491/task # for i in *; do echo -n "$i: " ; grep Name $i/status; done
3491: Name:     mediaserver
3576: Name:     mediaserver
3899: Name:     ApmTone
3900: Name:     ApmAudio
3901: Name:     ApmOutput
3905: Name:     FastMixer
3906: Name:     AudioOut_2
3909: Name:     soundTrigger cb
3910: Name:     Binder_1
3911: Name:     Binder_2
3912: Name:     Binder_3
3913: Name:     Binder_4
4302: Name:     Binder_5
4306: Name:     Binder_6
4328: Name:     Binder_7
4336: Name:     Binder_8
```

Layers::LibBinder

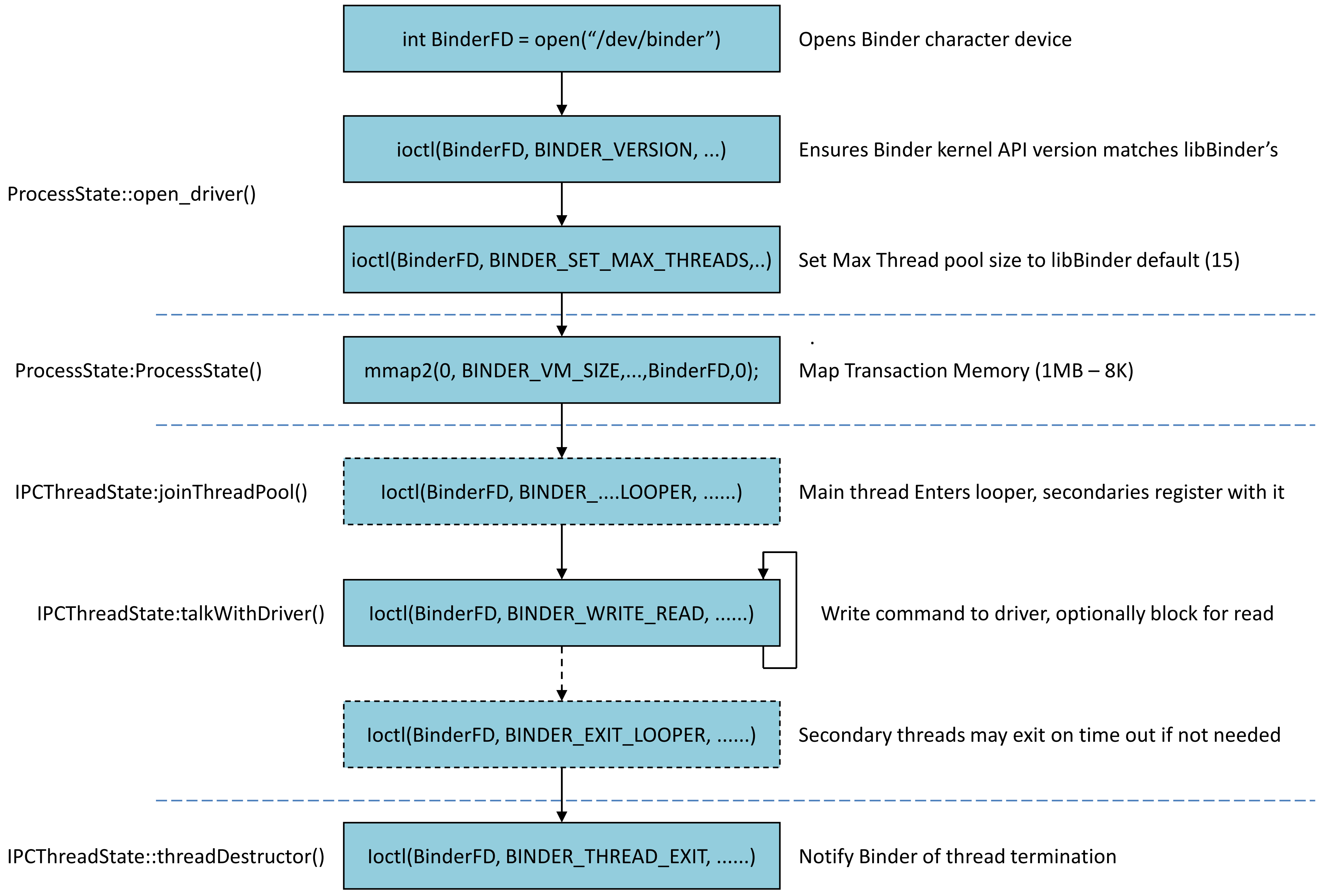
- Apps (read: Zygote) automatically start pool
  - frameworks/base/cmds/app_process/app_main.cpp

- 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

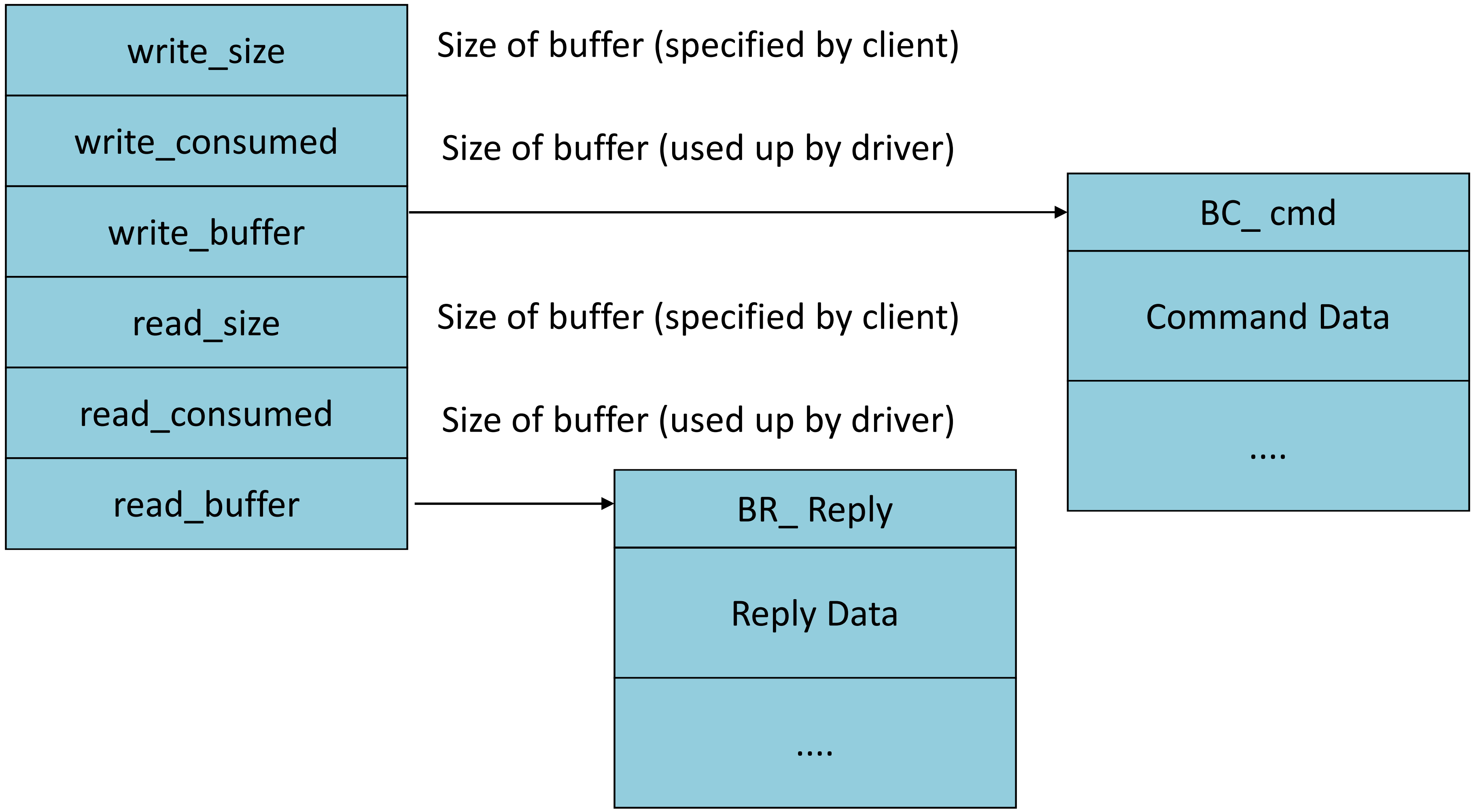
- All communication done via ioctl(2) calls
  - No read/write needed
  - Clients open, configure with ioctl(2) then either:
    - Enter a polling loop (IPCThreadState::setupPolling)
    - Block until message/timeout (IPCThread::joinThreadPool)


int BinderFD = open("/dev/binder")
Opens Binder character device
ioctl(BinderFD, BINDER_VERSION, ...)
Ensures Binder kernel API version matches libBinder's
ProcessState::open_driver()
ioctl(BinderFD, BINDER_SET_MAX_THREADS,..)
Set Max Thread pool size to libBinder default (15)
ProcessState:ProcessState()
mmap2(0, BINDER_VM_SIZE,...,BinderFD,0);
Map Transaction Memory (1MB – 8K)
IPCThreadState:joinThreadPool()
Ioctl(BinderFD, BINDER_....LOOPER, ...)
Main thread Enters looper, secondaries register with it
IPCThreadState:talkWithDriver()
Ioctl(BinderFD, BINDER_WRITE_READ, ...)
Write command to driver, optionally block for read
Ioctl(BinderFD, BINDER_EXIT_LOOPER, ...)
Secondary threads may exit on time out if not needed
IPCThreadState::threadDestructor()
Ioctl(BinderFD, BINDER_THREAD_EXIT, ...)
Notify Binder of thread termination

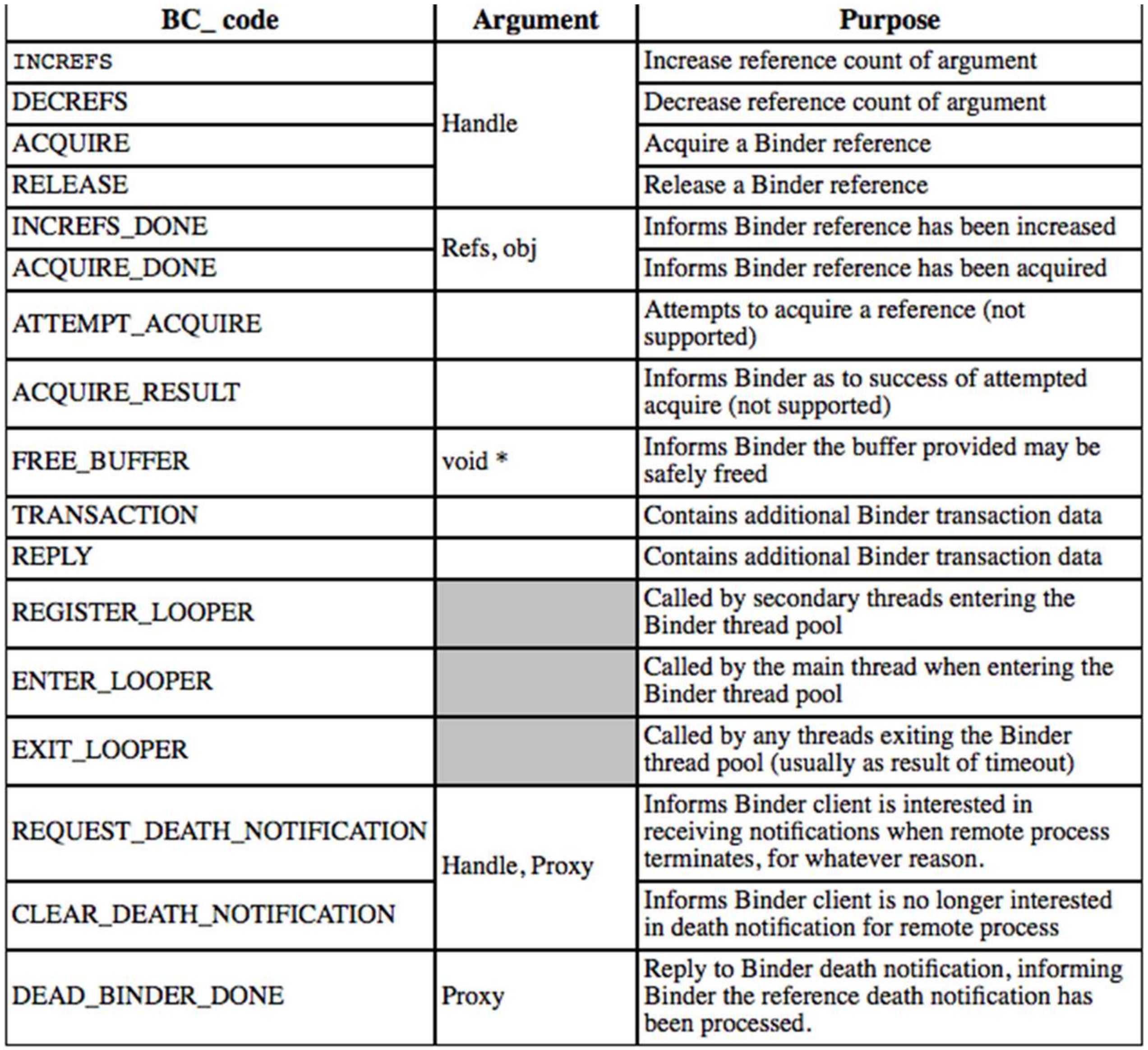
Layers::Kernel

- 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

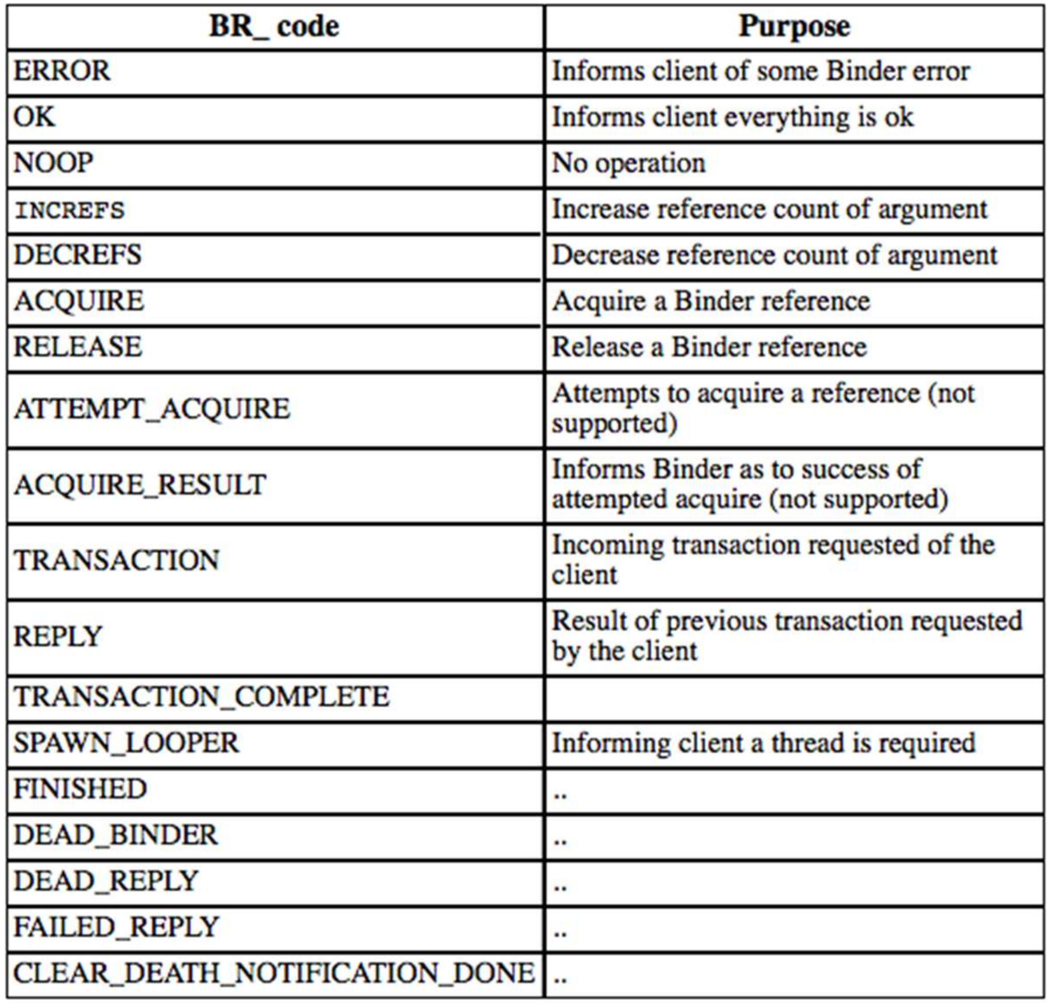

write_size
Size of buffer (specified by client)
write_consumed
Size of buffer (used up by driver)
write_buffer
BC_ cmd
Command Data
....
read_size
Size of buffer (specified by client)
read_consumed
Size of buffer (used up by driver)
read_buffer
BR_ Reply
Reply Data
....

Driver Protocol::Requests



BC_ code	Argument	Purpose
INCREFS	Handle	Increase reference count of argument
DECREFS		Decrease reference count of argument
ACQUIRE		Acquire a Binder reference
RELEASE		Release a Binder reference
INCREFS_DONE	Refs, obj	Informs Binder reference has been increased
ACQUIRE_DONE		Informs Binder reference has been acquired
ATTEMPT_ACQUIRE		Attempts to acquire a reference (not supported)
ACQUIRE_RESULT		Informs Binder as to success of attempted acquire (not supported)
FREE_BUFFER	void *	Informs Binder the buffer provided may be safely freed
TRANSACTION		Contains additional Binder transaction data
REPLY		Contains additional Binder transaction data
REGISTER_LOOPER	▨	Called by secondary threads entering the Binder thread pool
ENTER_LOOPER	▨	Called by the main thread when entering the Binder thread pool
EXIT_LOOPER	▨	Called by any threads exiting the Binder thread pool (usually as result of timeout)
REQUEST_DEATH_NOTIFICATION	Handle, Proxy	Informs Binder client is interested in receiving notifications when remote process terminates, for whatever reason.
CLEAR_DEATH_NOTIFICATION		Informs Binder client is no longer interested in death notification for remote process
DEAD_BINDER_DONE	Proxy	Reply to Binder death notification, informing Binder the reference death notification has been processed.

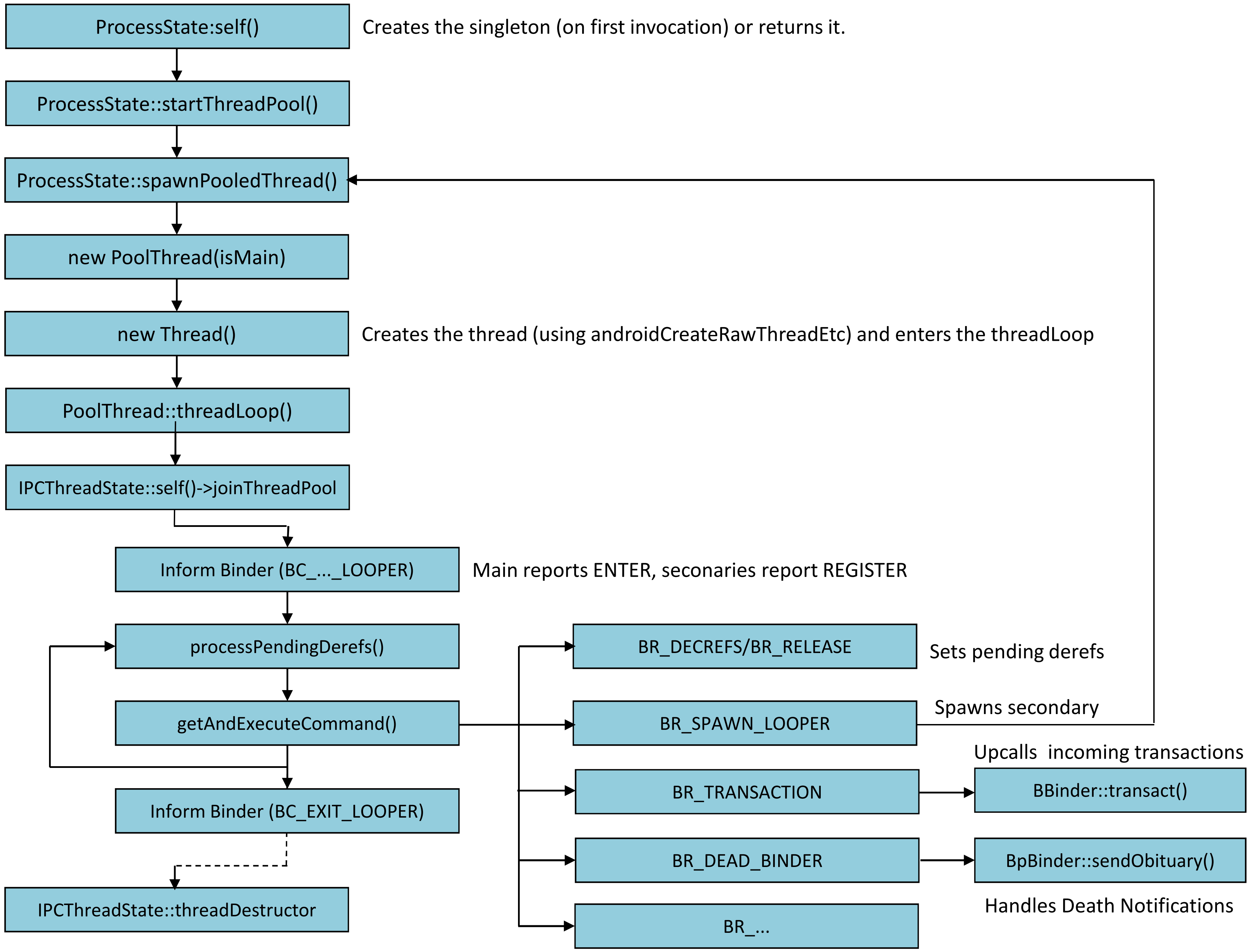
Driver Protocol::Replies



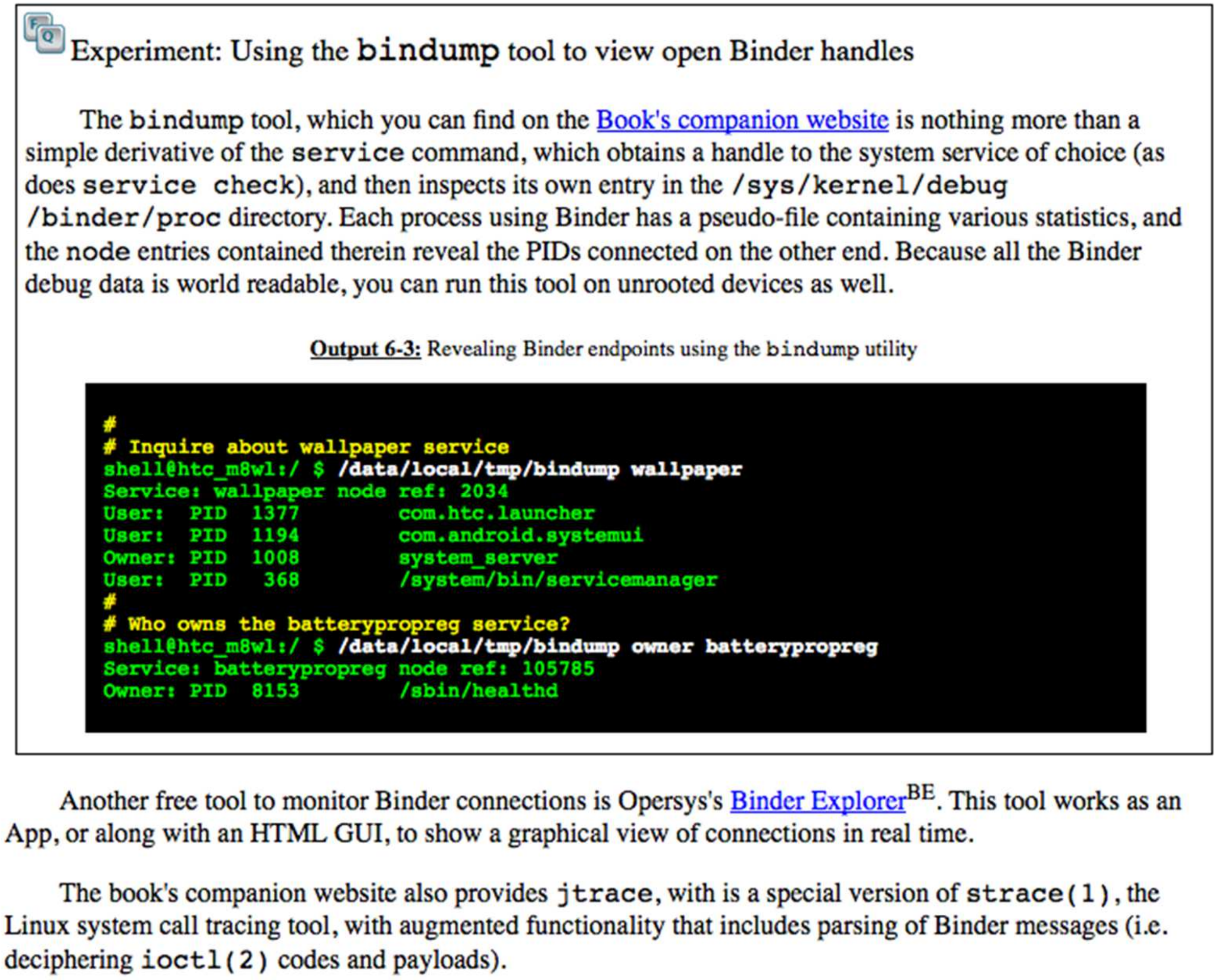
BR_ code	Purpose
ERROR	Informs client of some Binder error
OK	Informs client everything is ok
NOOP	No operation
INCREFS	Increase reference count of argument
DECREFS	Decrease reference count of argument
ACQUIRE	Acquire a Binder reference
RELEASE	Release a Binder reference
ATTEMPT_ACQUIRE	Attempts to acquire a reference (not supported)
ACQUIRE_RESULT	Informs Binder as to success of attempted acquire (not supported)
TRANSACTION	Incoming transaction requested of the client
REPLY	Result of previous transaction requested by the client
TRANSACTION_COMPLETE	
SPAWN_LOOPER	Informing client a thread is required
FINISHED	..
DEAD_BINDER	..
DEAD_REPLY	..
FAILED_REPLY	..
CLEAR_DEATH_NOTIFICATION_DONE	..

Binder Transaction Data

target	32-bit handle or pointer
cookie	Used to detect mismatched handles
code	Transaction code. One of the built-in codes, or application defined
flags	TF_ flags, indicating ONE_WAY, ACCEPT_FDS, or STATUS_CODE (ROOT_OBJECT unused)
sender_pid	Process identifier of sender
sender_uid	UID of message sender
data_size	If non zero, indicates data is a pointer to buffer of this number of bytes
offsets_size	If non zero, indicates data provides offsets into this message
data	Pointer to a buffer of data_size bytes, or offsets into message


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()
IPCThreadState::self()->joinThreadPool
Inform Binder (BC_..._LOOPER)
Main reports ENTER, seconaries report REGISTER
processPendingDerefs()
getAndExecuteCommand()
Inform Binder (BC_EXIT_LOOPER)
IPCThreadState::threadDestructor
BR_DECREFS/BR_RELEASE
Sets pending derefs
BR_SPAWN_LOOPER
Spawns secondary
Upcalls incoming transactions
BR_TRANSACTION
BBinder::transact()
BR_DEAD_BINDER
BpBinder::sendObituary()
Handles Death Notifications
BR_...

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 /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_m8wl:/ $ /data/local/tmp/bindump wallpaper
Service: wallpaper node ref: 2034
User:  PID  1377        com.htc.launcher
User:  PID  1194        com.android.systemui
Owner: PID  1008        system_server
User:  PID   368        /system/bin/servicemanager
#
# Who owns the batterypropreg service?
shell@htc_m8wl:/ $ /data/local/tmp/bindump owner batterypropreg
Service: batterypropreg node ref: 105785
Owner: PID  8153        /sbin/healthd
```

Another free tool to monitor Binder connections is Opersys's Binder Explorer[BE]. 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
  - Next training: February 8th, 2016, NYC!

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

Some References

Great discussion:

- http://events.linuxfoundation.org/images/stories/slides/abs2013_gargentas.pdf

Old, but nice:

- https://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf
- http://rts.lab.asu.edu/web_438/project_final/Talk%208%20AndroidArc_Binder.pdf

My book:

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