Explore Android Internals

Jim Huang (黃敬群) <**jserv**@0xlab.org>

Developer, **0xlab**

March 3, 2013 / Study-Area

Rights to copy

Corrections, suggestions, contributions and translations are welcome!

Latest update: Jan 10, 2014

Agenda

(0) Concepts

(1) Binder: heart of Android

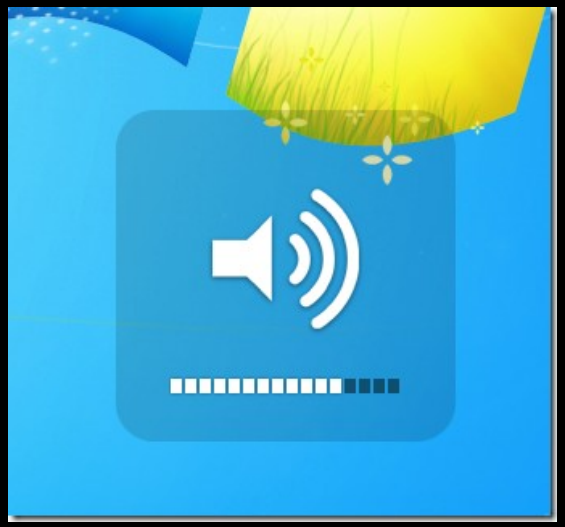
(2) Binder Internals

(3) System Services

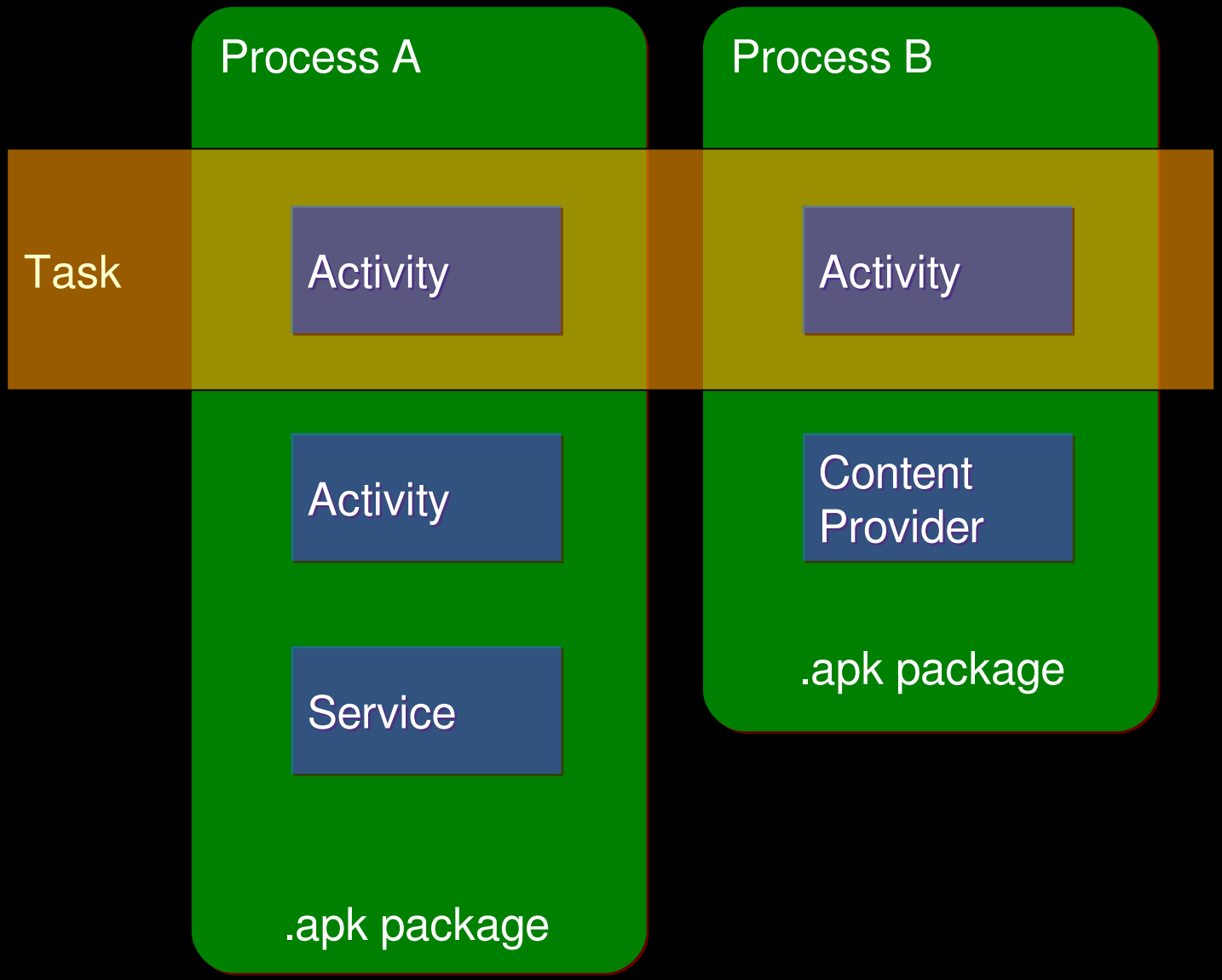
(4) Frameworks

Concepts

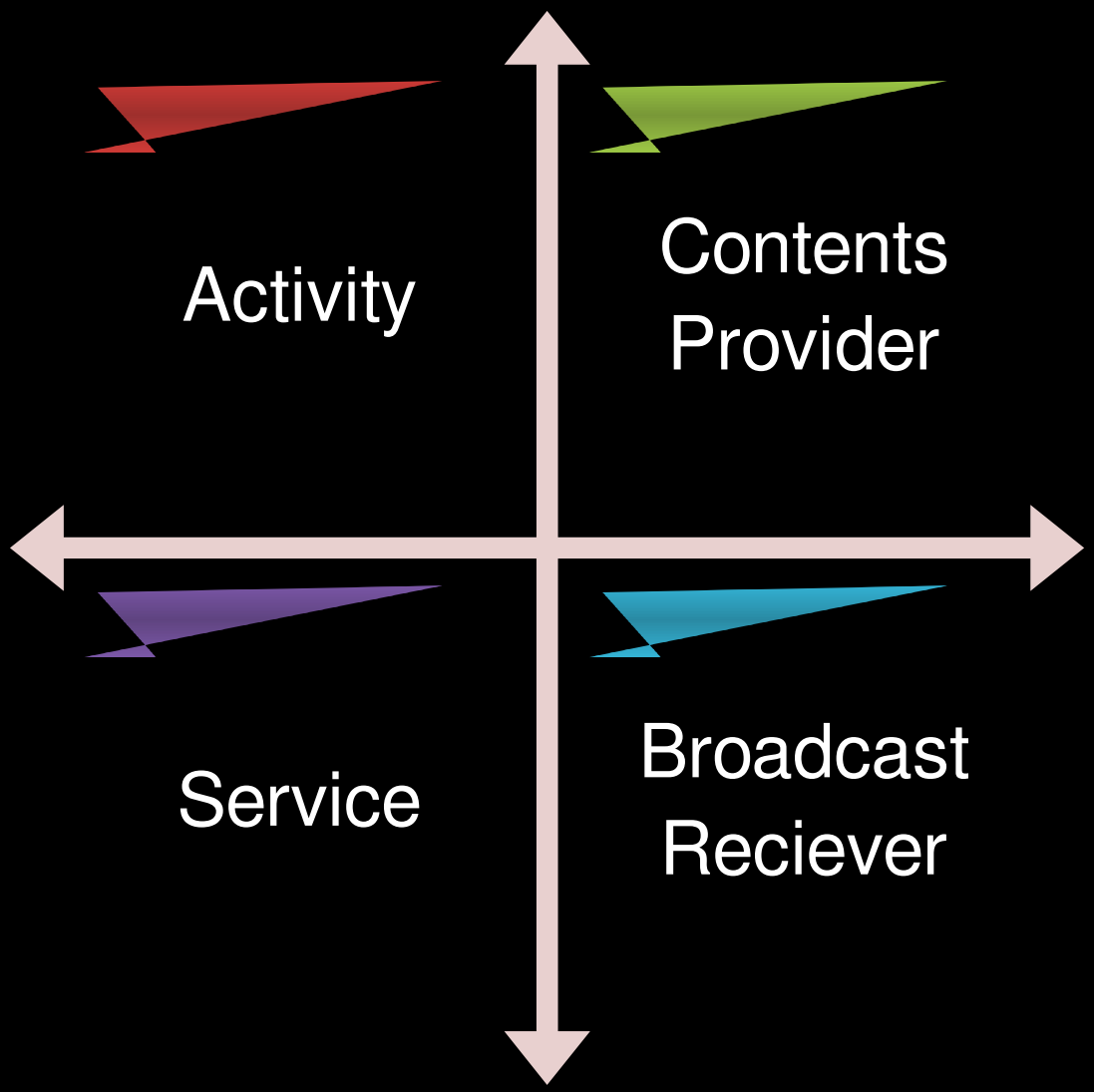
- Mobile environments → frequent Data communication among processes
  Example: mp3 player process → volume control process



Android Tasks

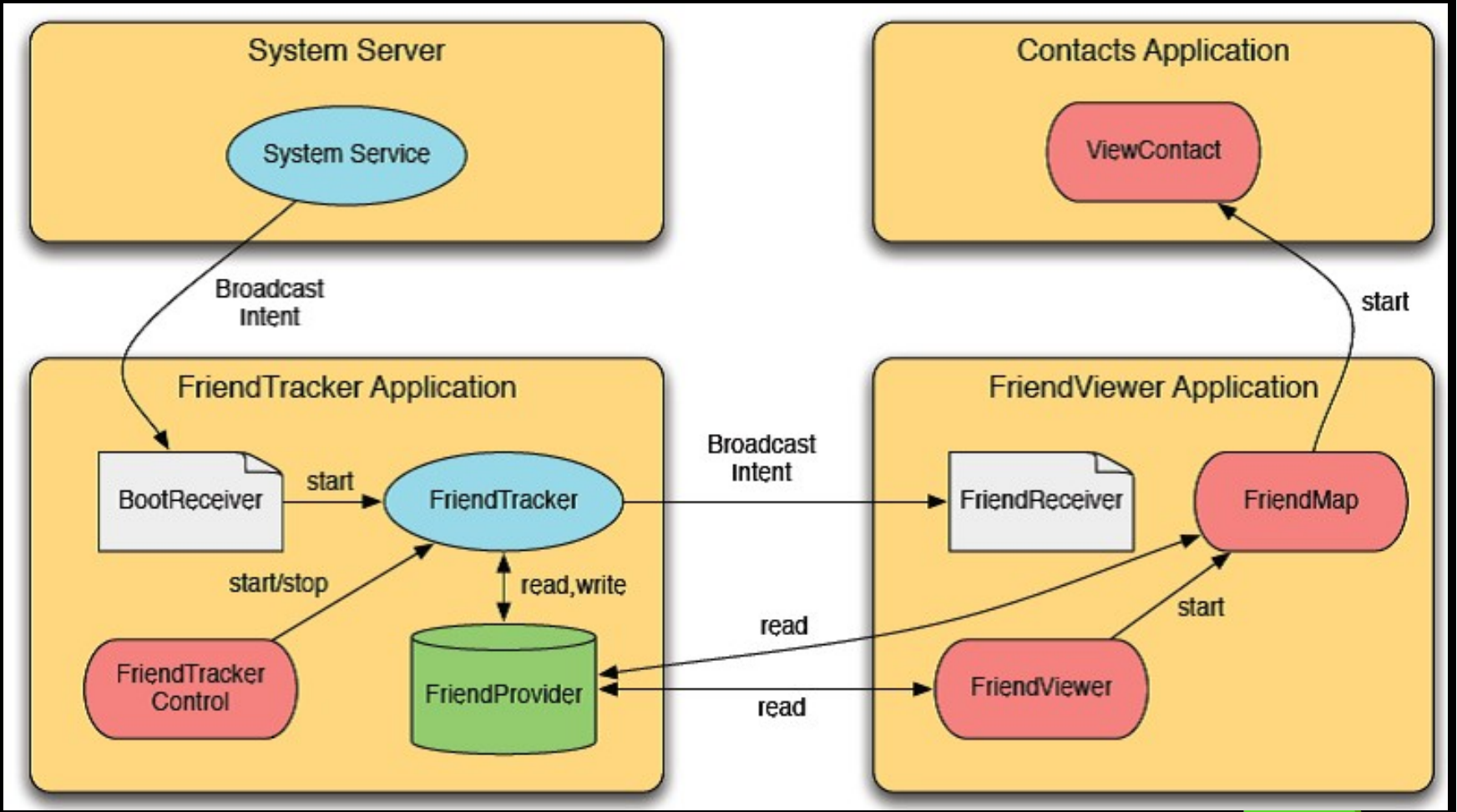


Our focus is the interaction among Android Activities/Services.


Activity
Contents
Provider
Service
Broadcast
Reciever

Component View

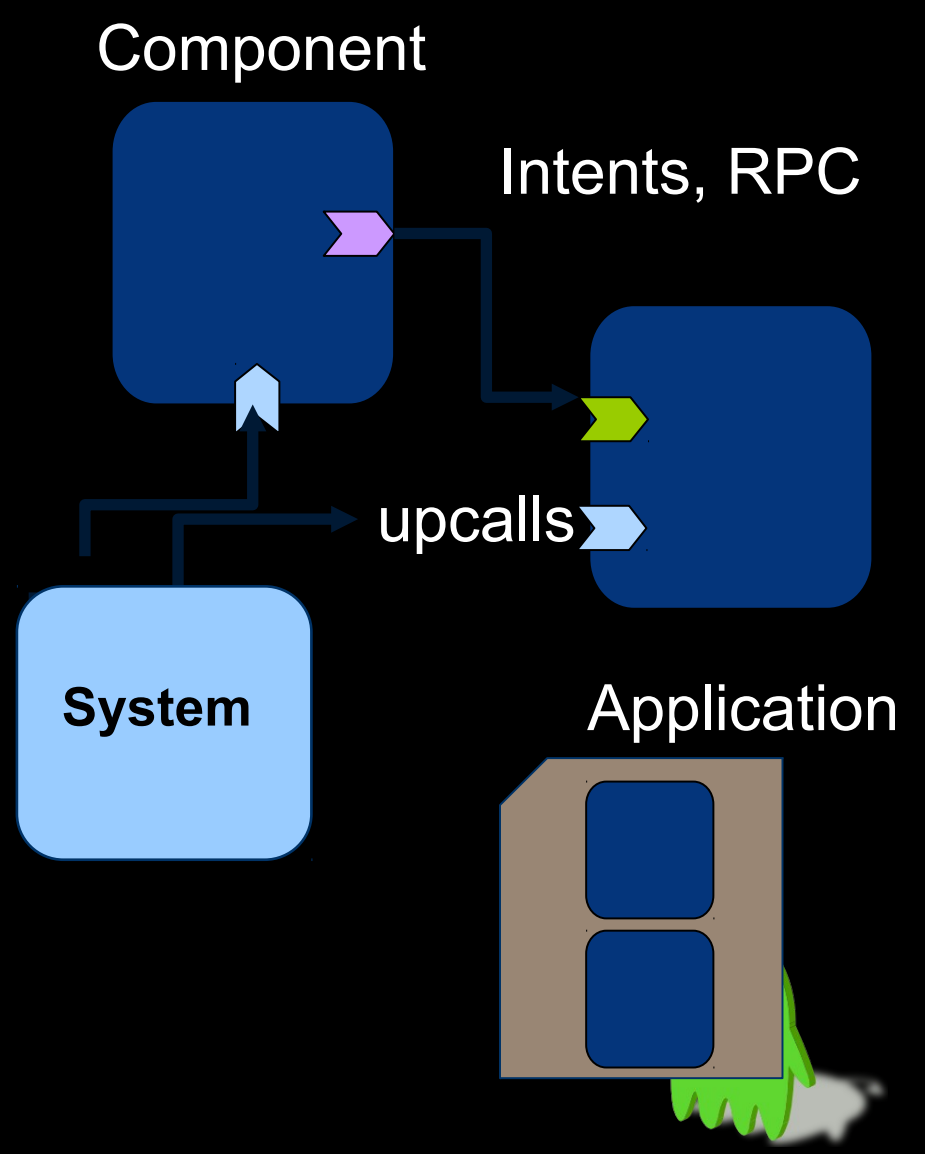
- Different component types
  - Activity
  - Service
  - Content Provider
  - Broadcast Receiver



Let's recall the behavior of Android Framework.

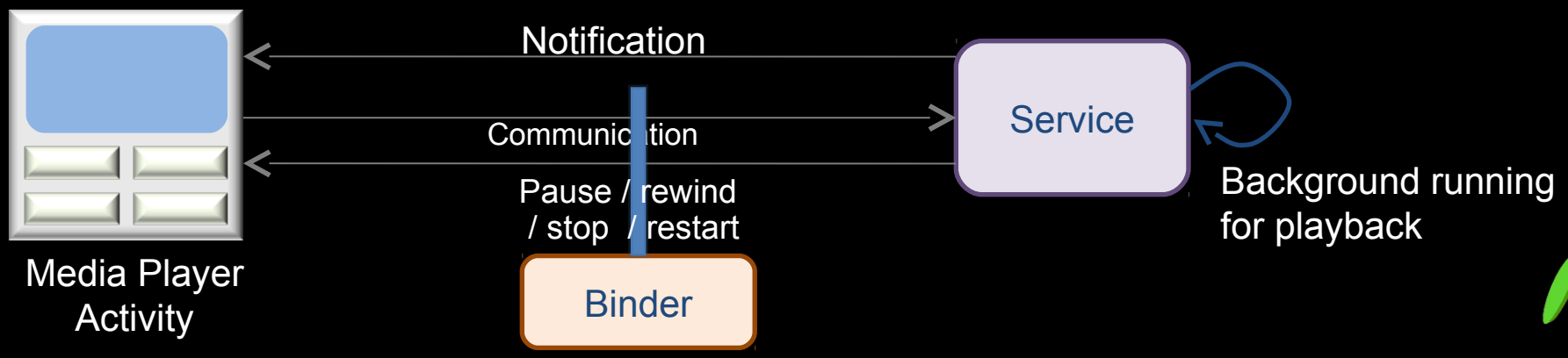
Android Components

- Applications declare typed components.
- metadata list of components (manifest)
- Components have upcall interfaces visible to system.
- System instantiates and destroys components driven by events in the system and UI.
- System upcalls components to notify them of lifecycle events.
- Applications may interact by typed messages among components.
- events (Intents)
- object invocation (Binder)



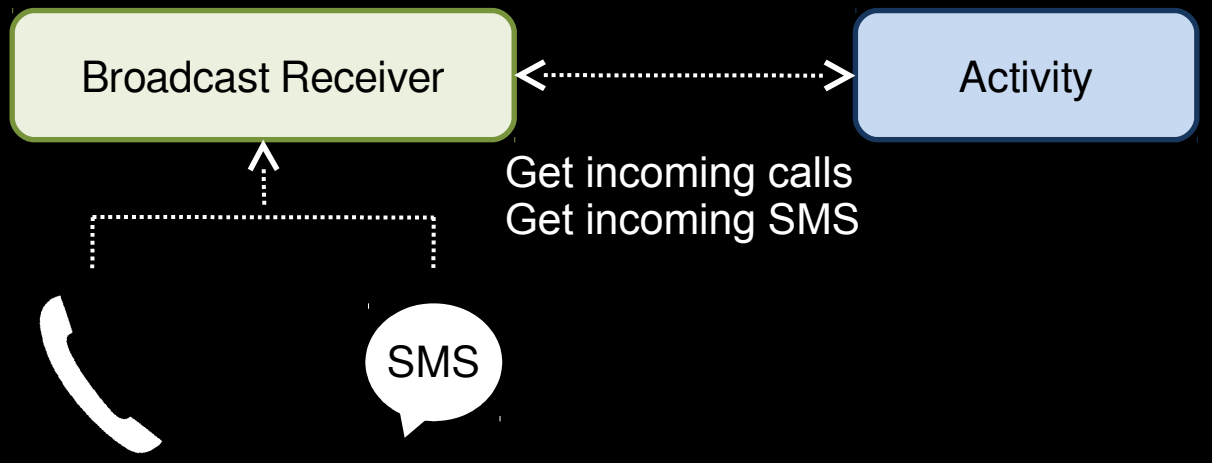
Components - Services

- A service does not have a visual user interface, but rather runs in the background for an indefinite period time.
  Example: music player, network download, etc
- Each service extends the Service base class.
- It is possible to bind to a running service and start the service if it's not already running.
- While connected, it is possible communicate with the service through an interface defined in AIDL.



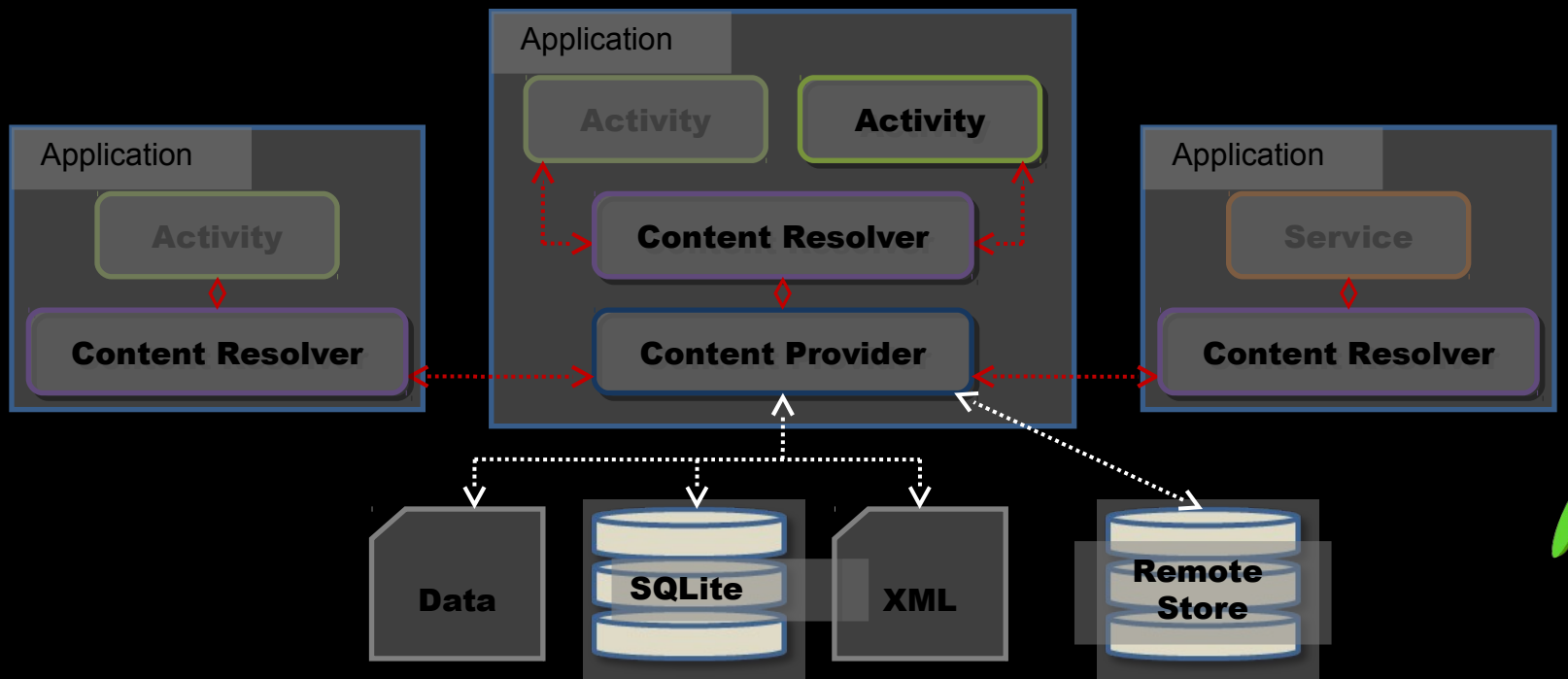
Components - Broadcast Receiver

- A broadcast receiver is a component that receives and reacts to broadcast announcements (Intents).
  - Many broadcasts originate in system code.
    - Example: announcements that the time zone has changed, that the battery is low, etc.
  - Applications can also initiate broadcasts.
    - Example: to let other applications know that some data has been downloaded to the device and is available for them to use.
- All receivers extend the **BroadcastReceiver** base class.

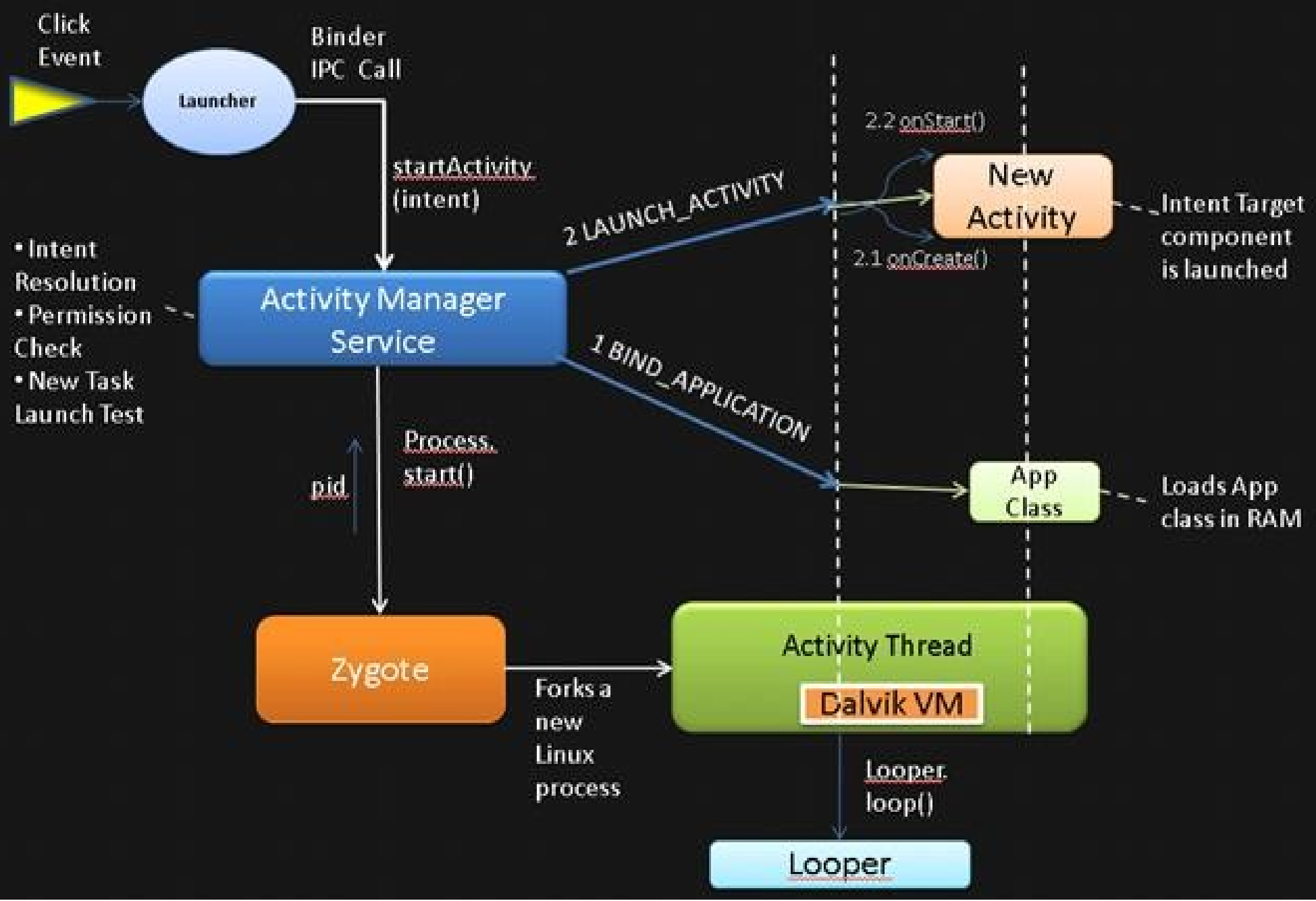


Components - Content Provider

- A content provider makes a specific set of the application's data available to other applications.
  - The data can be stored in the file system, in an SQLite, or in any other manner that makes sense.
- Using a content provider is the only way to share data between Android applications.



Application Launch



Zygote

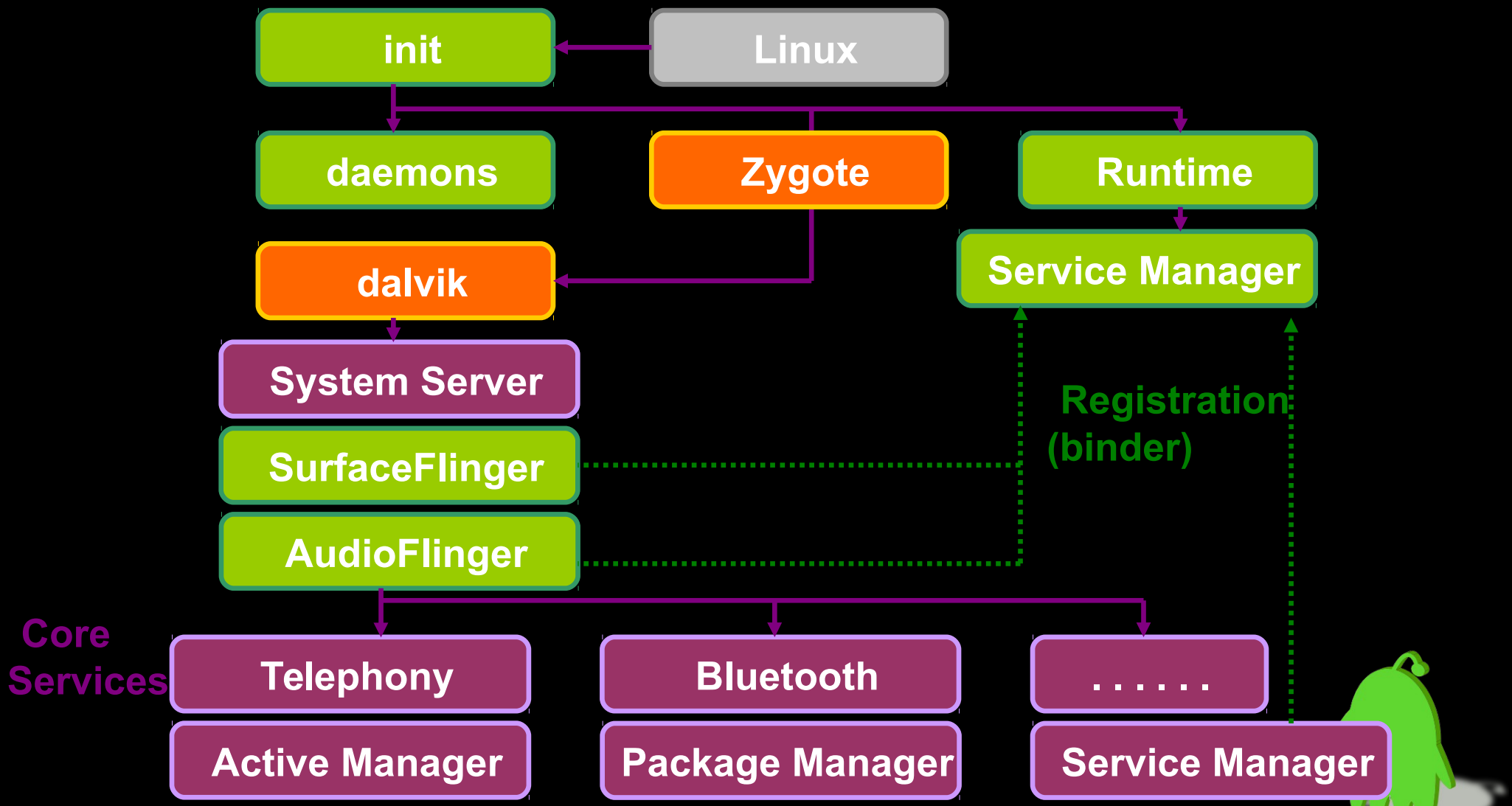
- a preforked simple process that the system keeps around that makes new application startup faster.
- It sits idle in a neutral state (that is, "unfertilized" by a specific application) until it's needed, at which point the system instructs it to exec() the appropriate application.
  - A new zygote process is then started up in the background to replace the old, waiting for the next process to start.
  - You absolutely can not create any threads in zygote, since it will be forking (without exec) from itself.

`$> chrome "--renderer-cmd-prefix=gdb -args"`

→ Using the `--renderer-cmd-prefix` option bypasses the zygote launcher.

Zygote & boot sequence

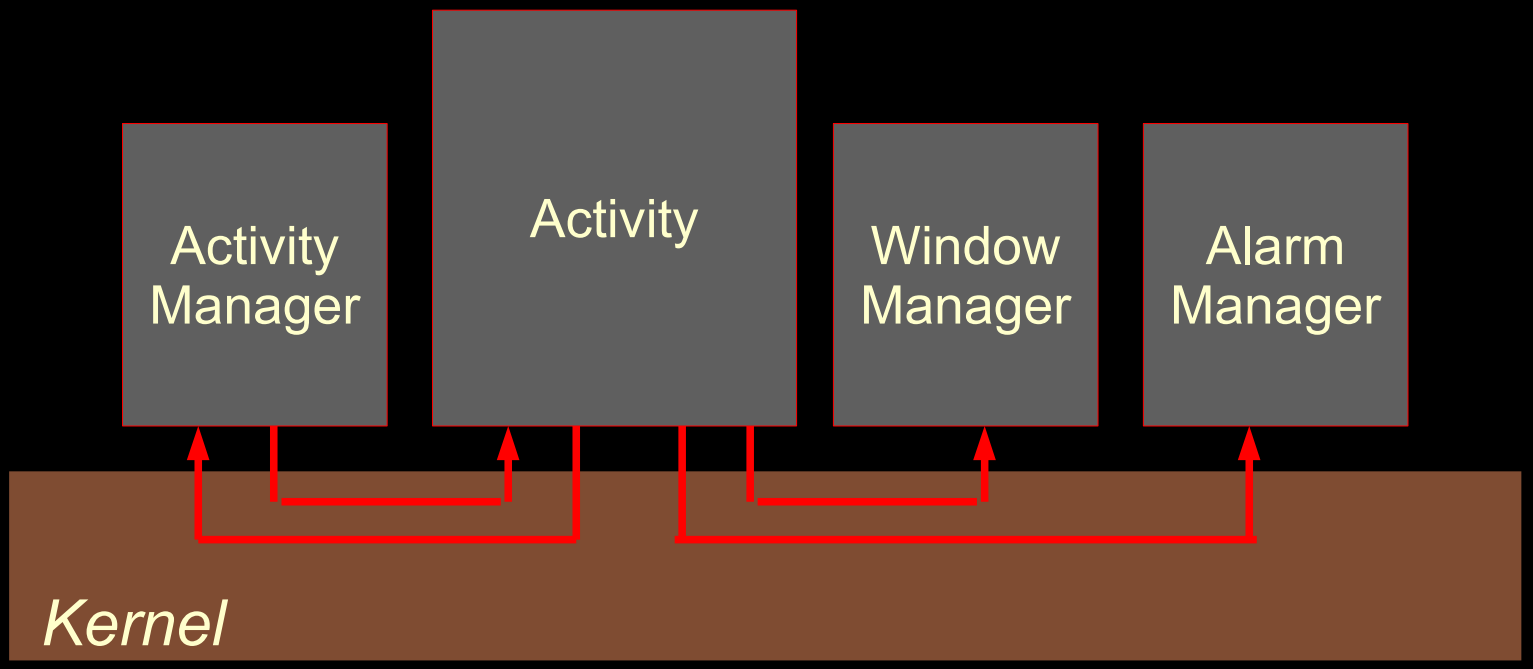
- System Server starts the core Android services.
  ActivityManager, WindowManager, PackageManager, etc



Binder IPC/RPC: The heart of Android

NOTE: This presentation only covers Android 4.0

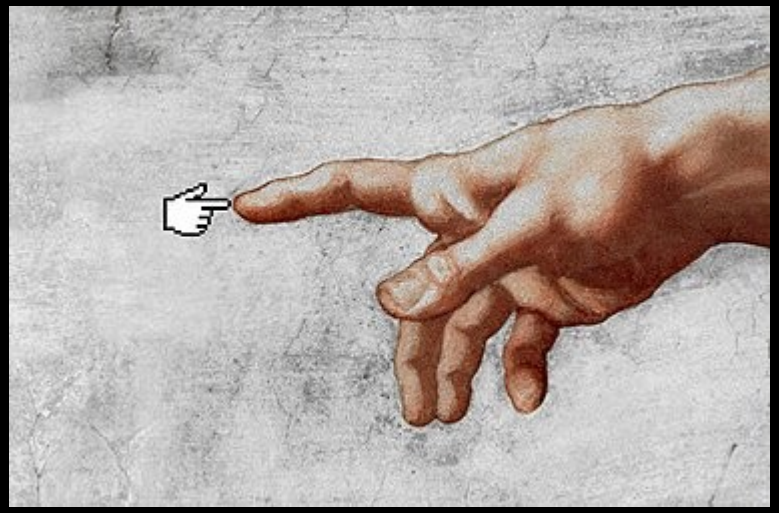
IPC = Inter-Process Communication



Let's get back to Binder IPC. What is the essential working model?

Why IPC?

- Each process has its own address space
- Provides data isolation
- Prevents harmful direct interaction between two different processes
    - Sometimes, communication between processes is required for modularization

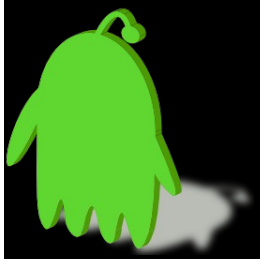


IPC Mechanisms

- In GNU/Linux
  - Signal
  - Pipe
  - Socket
  - Semaphore
  - Message queue
  - Shared memory
- In Android
  - Binder: lightweight RPC (Remote Procedure Communication) mechanism

Binder History

- Developed under the name OpenBinder by Palm Inc. under the leadership of Dianne Hackborn
- Android Binder: customized and reduced re-implementation of OpenBinder, providing bindings to functions/data from one execution env to another



OpenBinder

- A complete open-source solution supporting new kinds of component-based system-level design.
- Resource management between processes.
- Binder is system oriented rather than application oriented.
- Support a new kind of component-based system-level development.
- Component/object representations of various basic system services.
- Binder itself does not impose a particular threading model. The Binder has been used to implement a wide variety of commercial-quality system-level services.

openbinder

palmsource OHA

Thread Priority binding to OpenBinder

[prio of thread creation in kernel]

- Do you need Realtime Characterestics for Mobile?
- Why does nobody talked about the characteristics of real-time kernel androids not?
- We can use preemptible kernel in kernel space for realtime, but how about userspace realtime?
- We can implement userspace realtime application with Locked Mutex Method (FUTEX)
  - Example: Priority Queuing , Priority Inheritance , Robust mutex

Throughput and Latency!

Background Problems

- Applications and Services may run in separated processes but must communicate and share data
- Android natively supports a **multi-threading** environment.
- An Android application can be composed of multiple *concurrent* threads.
- How to create threads in Android? Like in Java!

D-Bus does suffer from such issues if socket backend is used.

Binder: Android's Solution

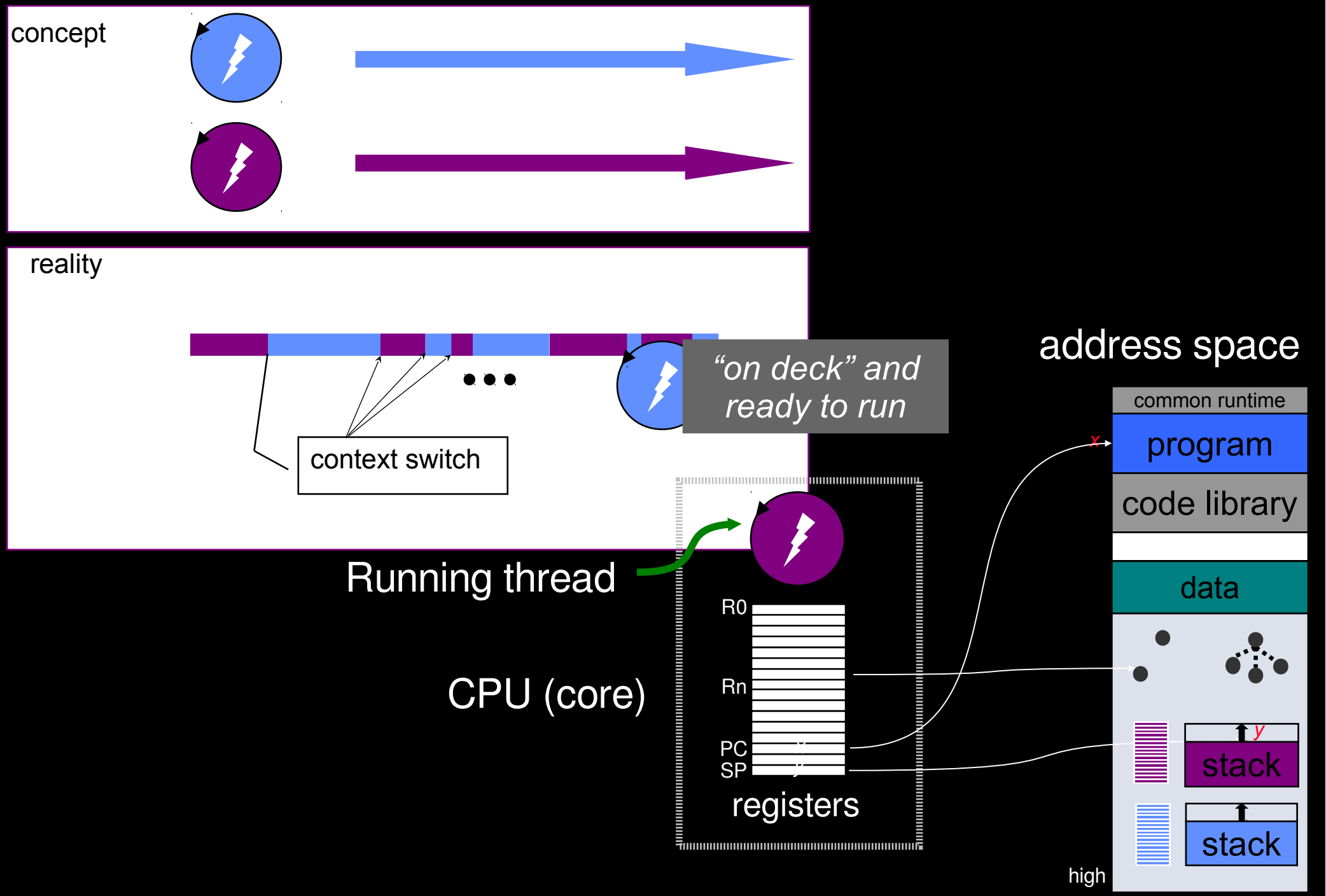
- Driver to facilitate inter-process communication
- High performance through shared memory
- Per-process thread pool for processing requests
- Reference counting, and mapping of object references across processes
- Synchronous calls between processes

"In the Android platform, the binder is used for nearly everything that happens across processes in the core platform. " – Dianne Hackborn
https://lkml.org/lkml/2009/6/25/3

Binder: Development background

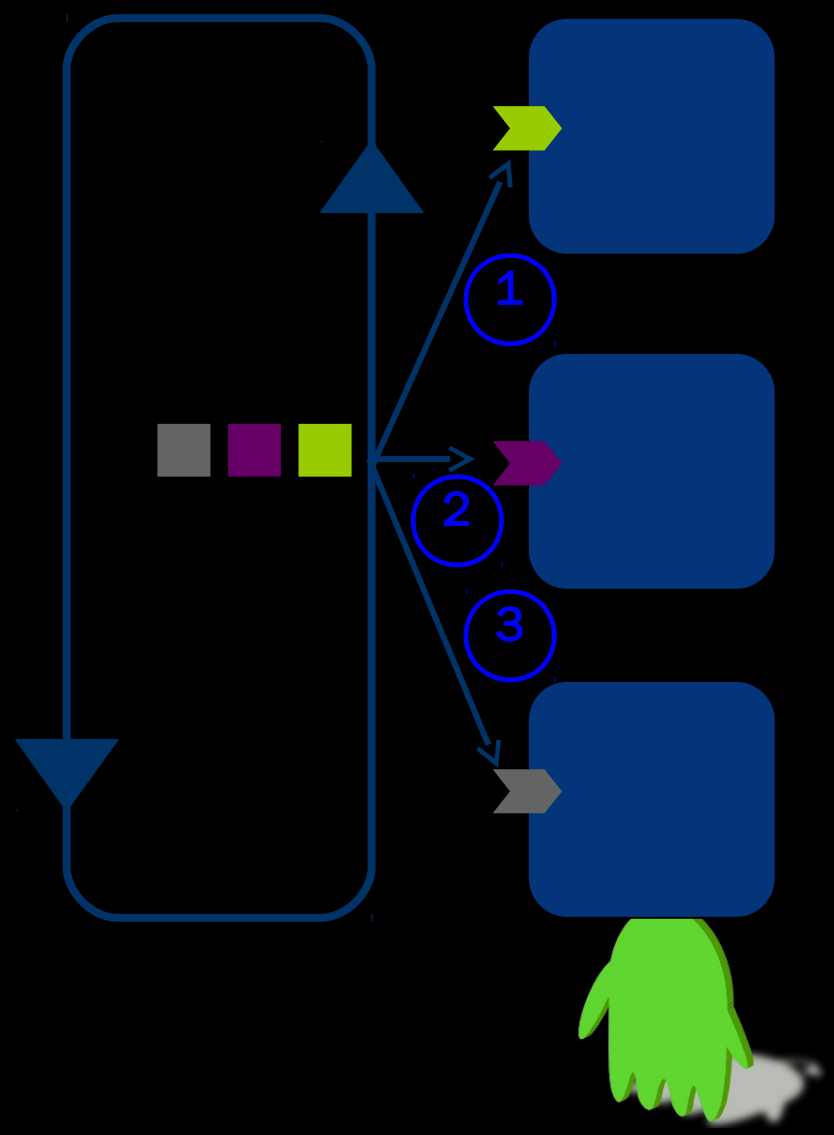
- The model of Binder was created for a microkernel-like device (BeOS).
- very limited, inflexible in its use-cases, but very powerful and extremely low-overhead and fast.
- Ensure that the same CPU timeslice will go from the calling process into the called process's thread, and then come back into the caller when finished.
  - → There is almost no scheduling involved, and is much like a syscall into the kernel that does work for the calling process.
  - → This interface is very well suited for cheap devices with almost no RAM and very low CPU resources.

Threads in Real environments, Always!



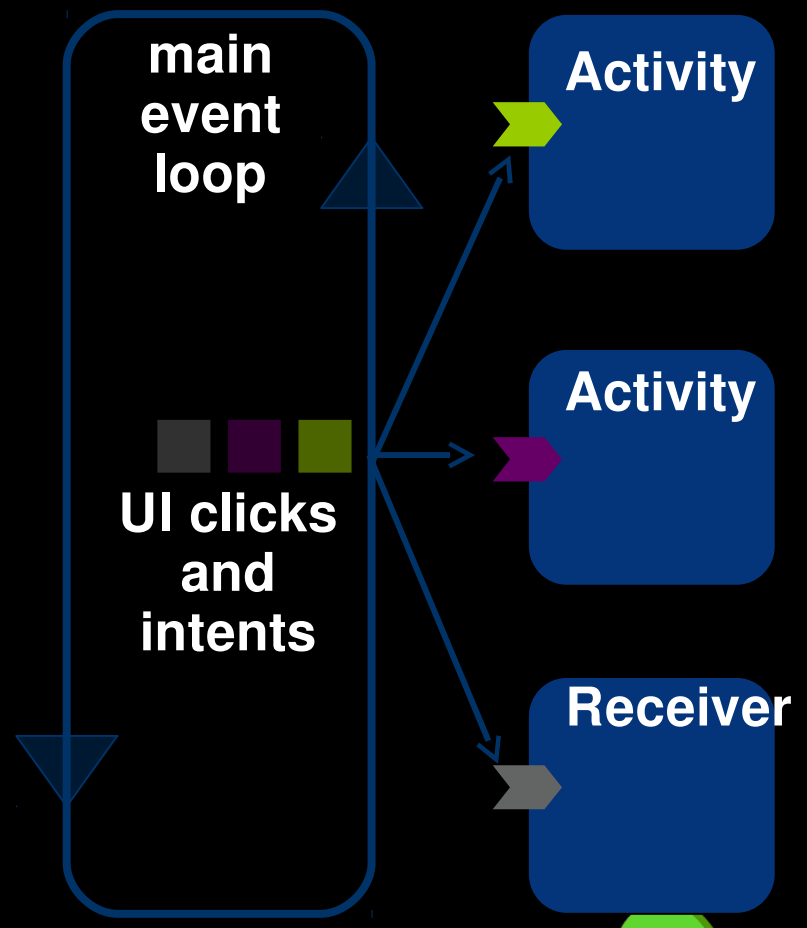
Android Apps: main event loop

- The main thread of an Android application is called the Activity Thread.
- It receives a sequence of events and invokes their handlers.
- Also called the “UI thread” because it receives all User Interface events.
  - screen taps, clicks, swipes, etc.
  - All UI calls must be made by the UI thread: the UI lib is not thread-safe.
  - MS-Windows apps are similar.
- The UI thread must not block!
  - If it blocks, then the app becomes unresponsive to user input: bad.



A closer look of main event loop

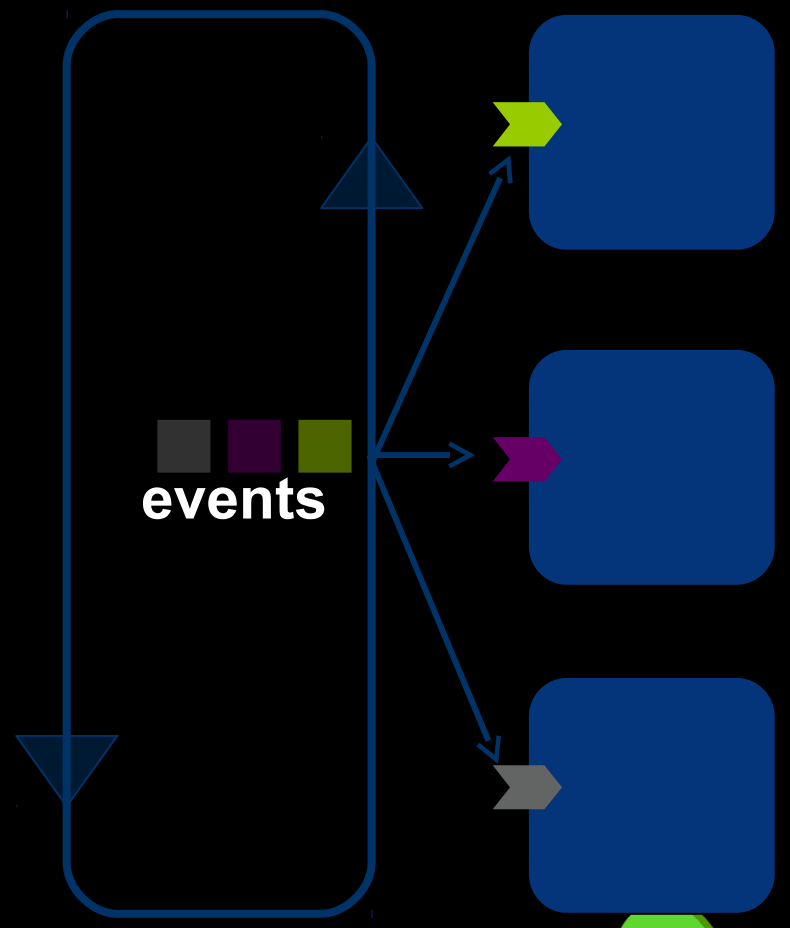
- The main thread delivers UI events and intents to Activity components.
- It also delivers events (broadcast intents) to Receiver components.
- Handlers defined for these components must not block.
- The handlers execute serially in event arrival order.
- Note: Service and ContentProvider components receive invocations from other apps (i.e., they are servers).
- These invocations run on different threads...more on that later.



Dispatch events by invoking component-defined handlers.

Event-driven programming

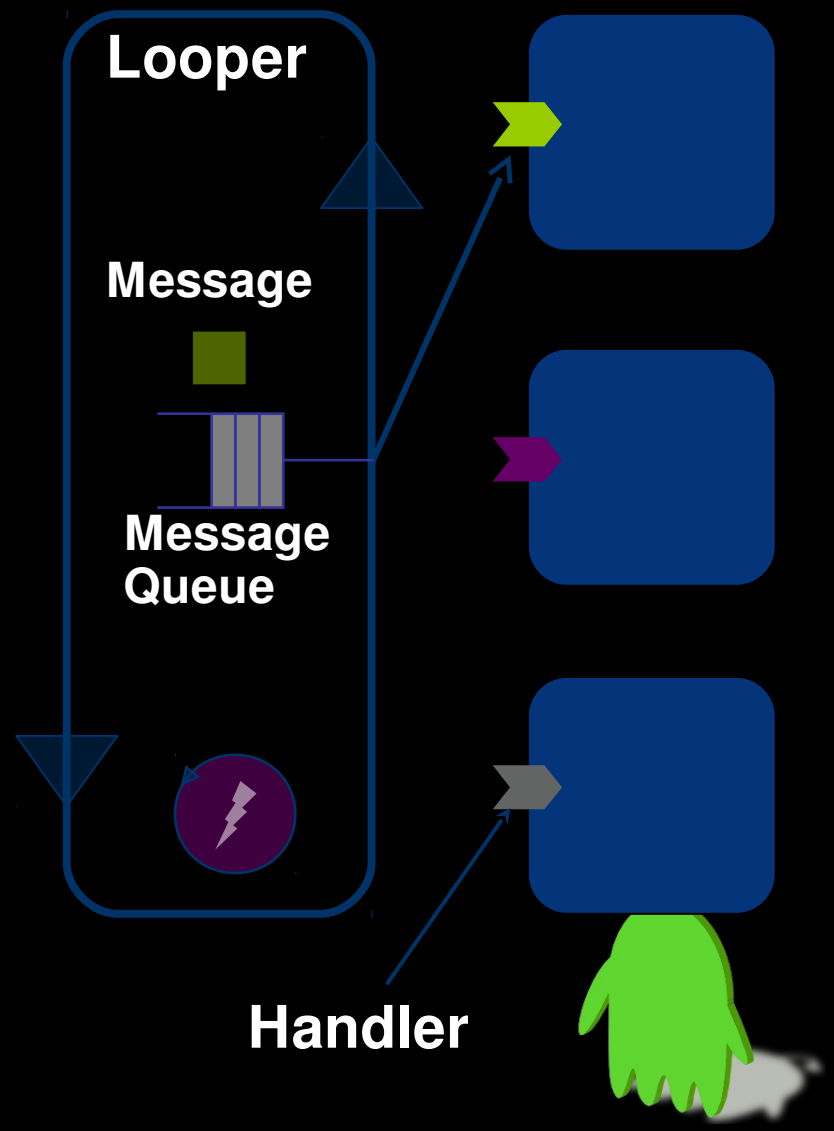
- This “design pattern” is called event-driven (event-based) programming.
- In its pure form the thread never blocks, except to wait for the next event, whatever it is.
- We can think of the program as a set of handlers: the system upcalls a handler to dispatch each event.
- Note: here we are using the term “event” to refer to any notification:
  - arriving input
  - asynchronous I/O completion
  - subscribed events
  - child stop/exit, “signals”, etc.



Dispatch events by invoking handlers (upcalls).

Android Events

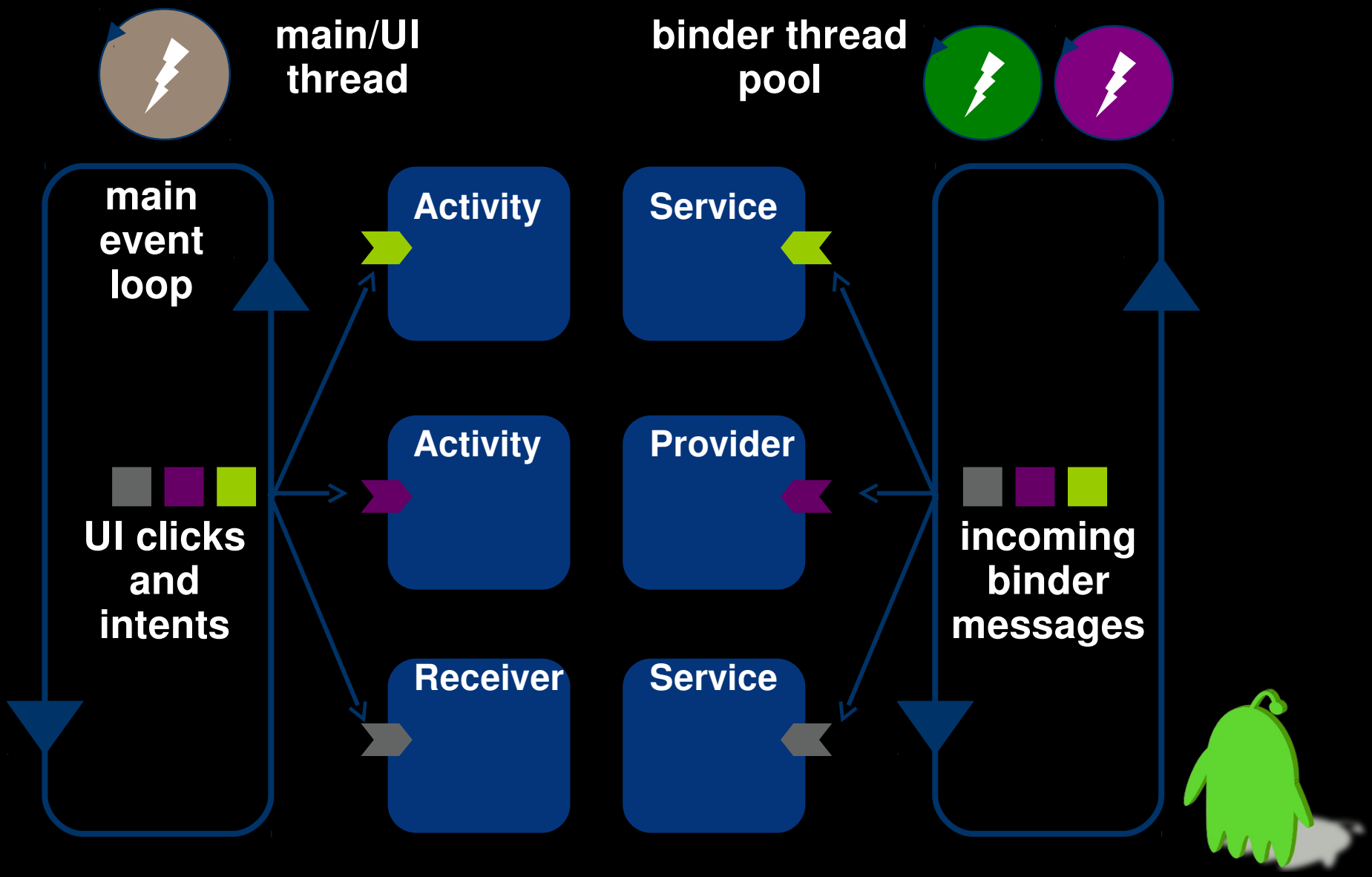
- Android defines a set of classes for event-driven programming in conjunction with threads.
- A thread may have at most one Looper bound to a MessageQueue.
- Each Looper has exactly one thread and exactly one MessageQueue.
- The Looper has an interface to register Handlers.
- There may be any number of Handlers registered per Looper.
- These classes are used for the UI thread, but have other uses as well.



Handler & Looper

- A Handler allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Handlers bind themselves to a thread context. If a Handler is created with its empty constructor, it is bound to the calling thread.
- A Looper used to run a message loop for a thread. Threads by default do not have a message loop associated with them; to create one, call prepare() in the thread that is to run the loop, and then loop() to have it process messages until the loop is stopped.

Android: adding services



Pool of Event-Driven Threads

- Android Binder receives a sequence of events (intents) in each process.
- They include incoming intents on provider and service components.
- Handlers for these intents may block. Therefore the app lib uses a pool of threads to invoke the Handlers for these incoming events.
- Many Android Apps don't have these kinds of components: those Apps can use a simple event-driven programming model and don't need to know about threads at all.
- But Apps having these component types use a different design pattern: pool of event-driven threads.
- This pattern is also common in multi-threaded servers, which poll socket descriptors listening for new requests.

Multi-threaded RPC server

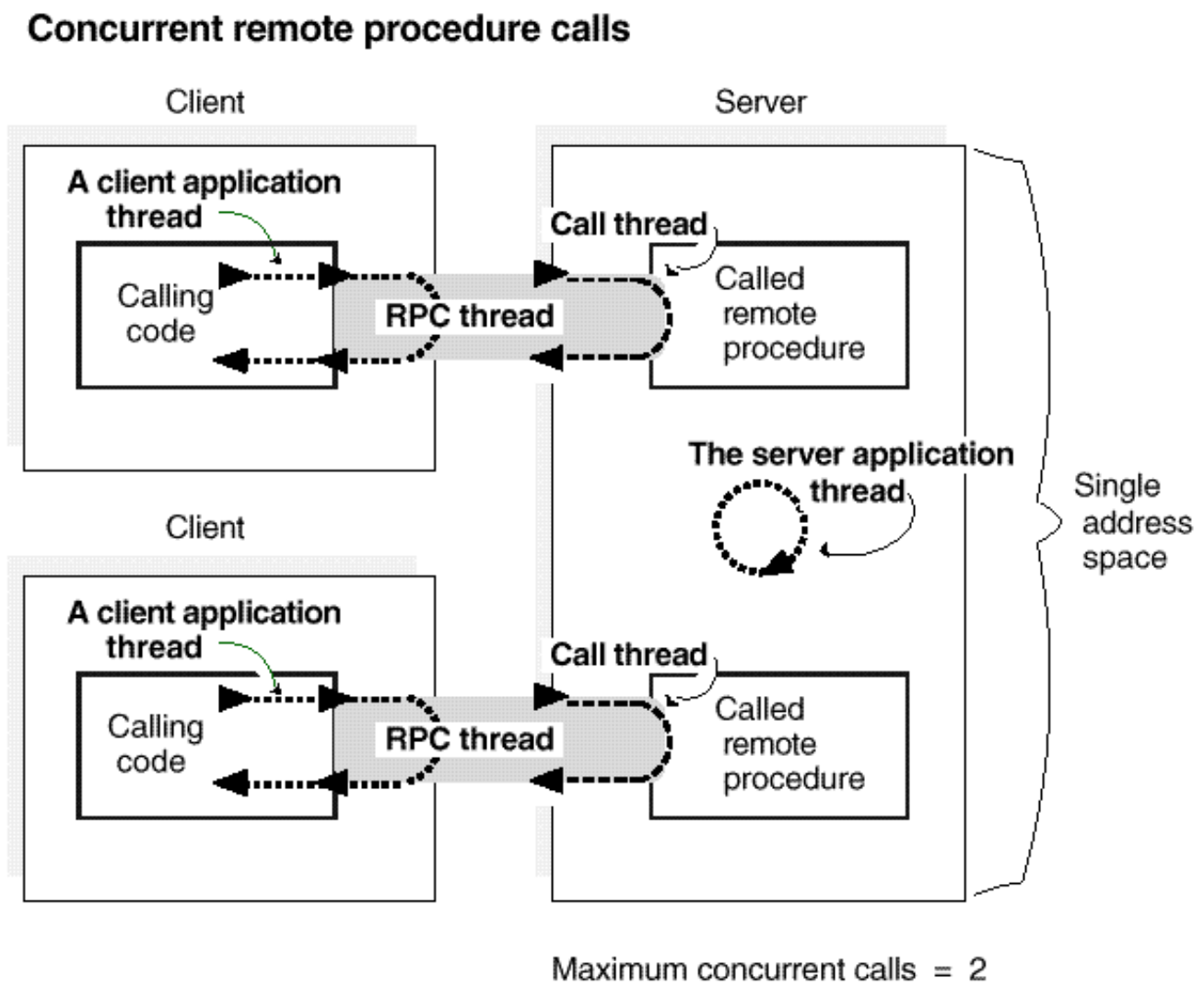


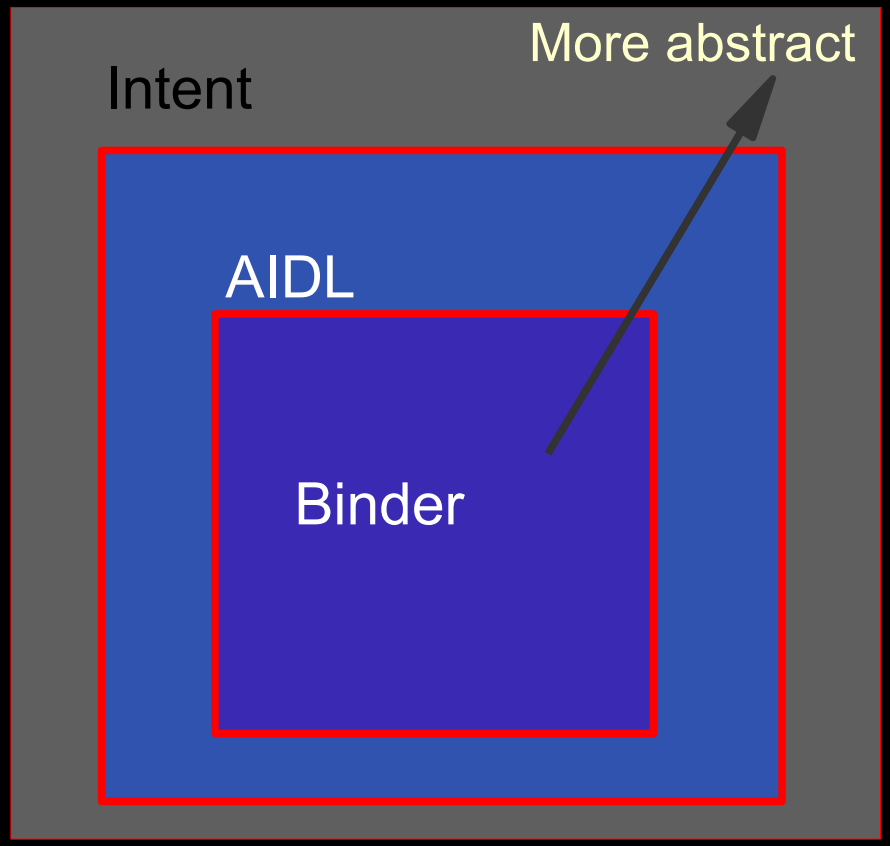
Figure 6-2 Concurrent Call Threads Executing in Shared Execution Context

[OpenGroup, late 1980s]

Binder & Zygote

- Each child of Zygote contains the Binder runtime. This runtime contains a small thread pool that block on Binder's kernel driver to handle requests.
  - → Binder threads only handle Binder requests.
  - → Binder requests are directly sent by the calling thread using Binder's (blocking) **`transact`** method to perform a remote method call. (Proxy - **`BinderProxy.transact`**, Service – **`Binder.transact`**)
  - → To switch thread contexts use Handlers and Loopers.

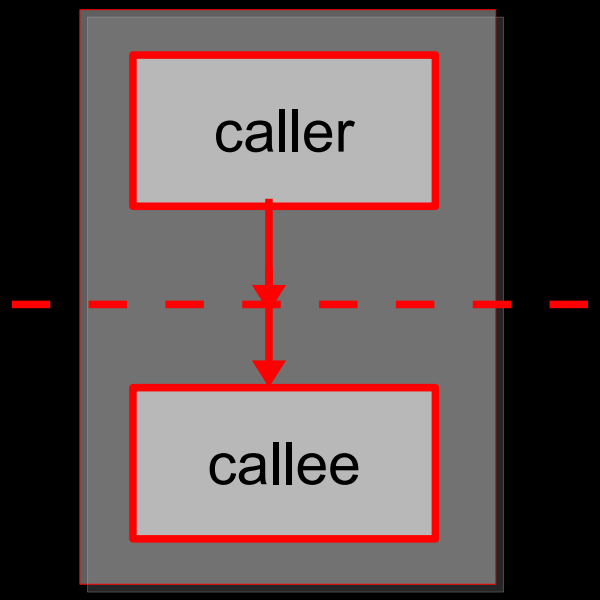
IPC Abstraction



- Intent
  The highest level abstraction
- Inter process method invocation
  AIDL: Android Interface Definition Language
- binder: kernel driver
- ashmem: shared memory

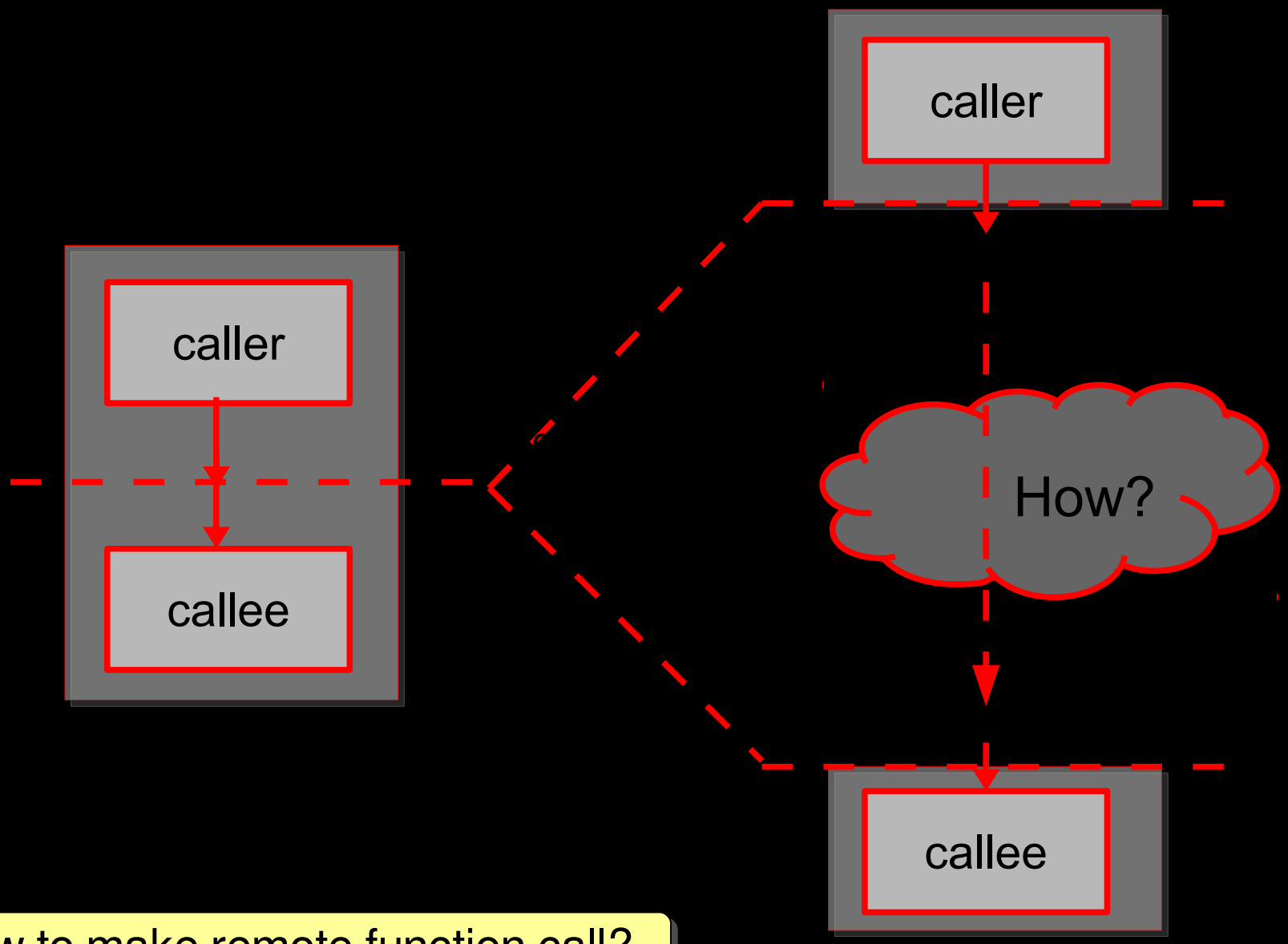
Level of abstraction: Binder → AIDL → Intent

Method invocation



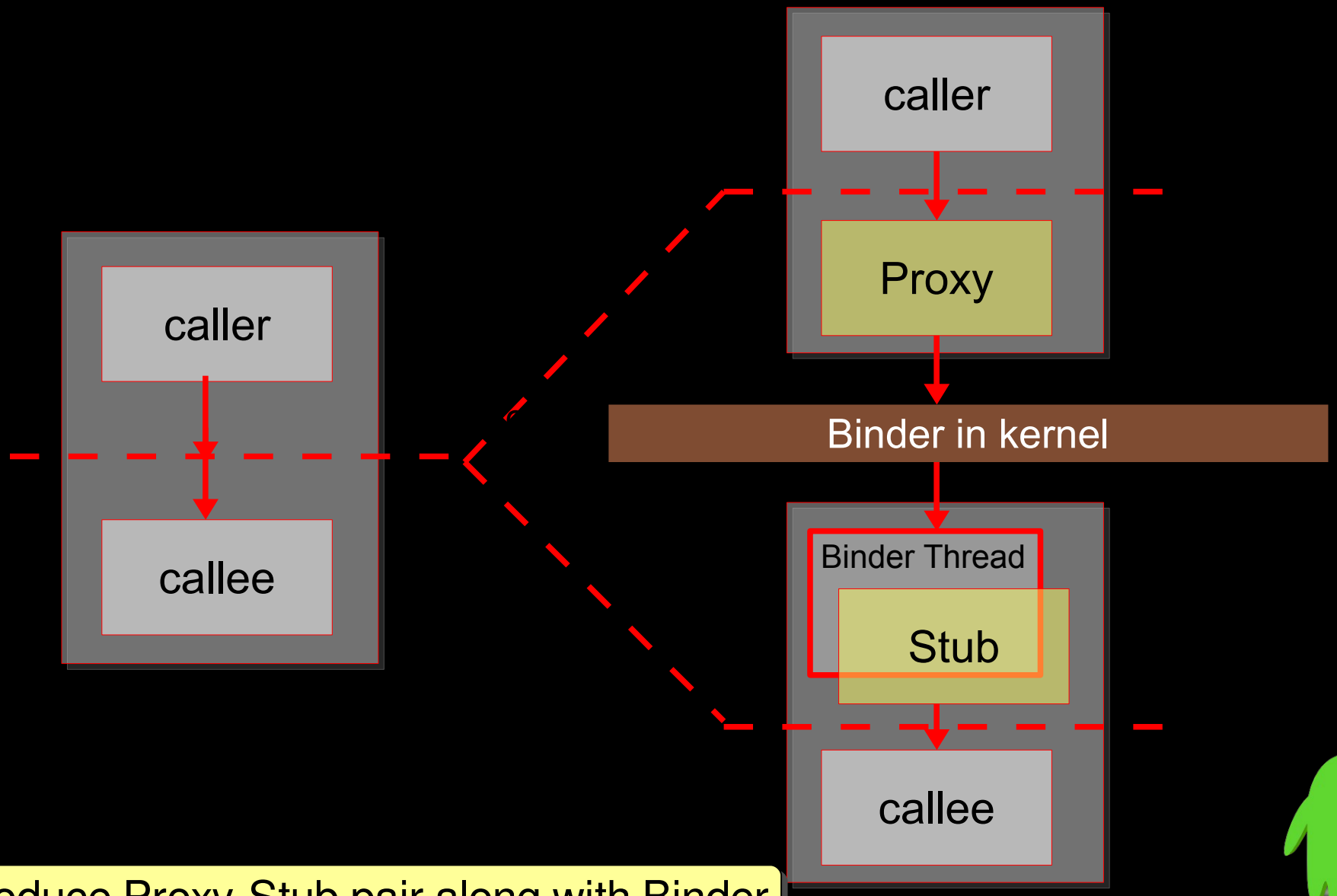
Think of how the typical function call works:
caller (call somebody) + callee (somebody called)

Inter-process method invocation

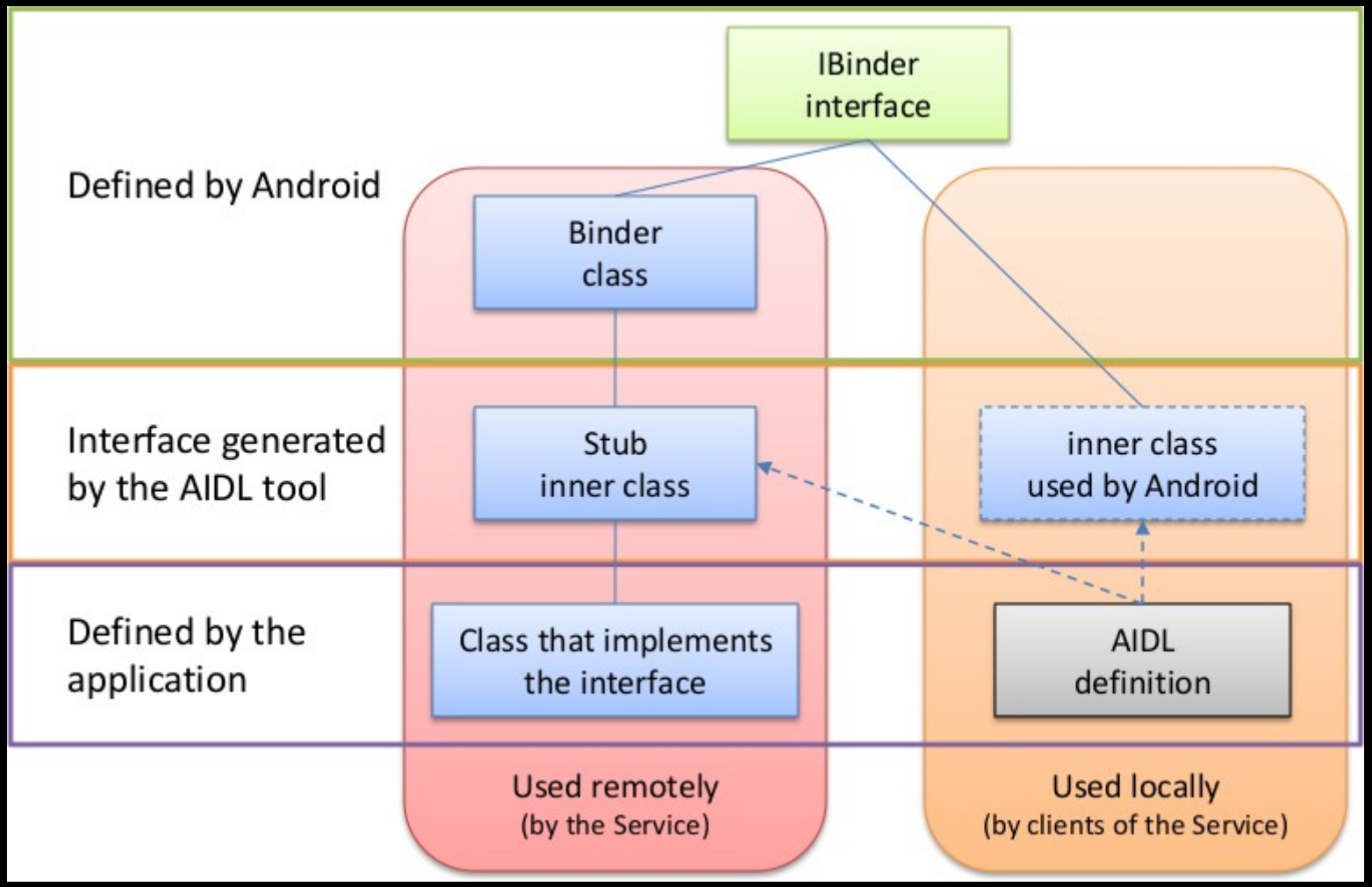


How to make remote function call?

Inter-process method invocation

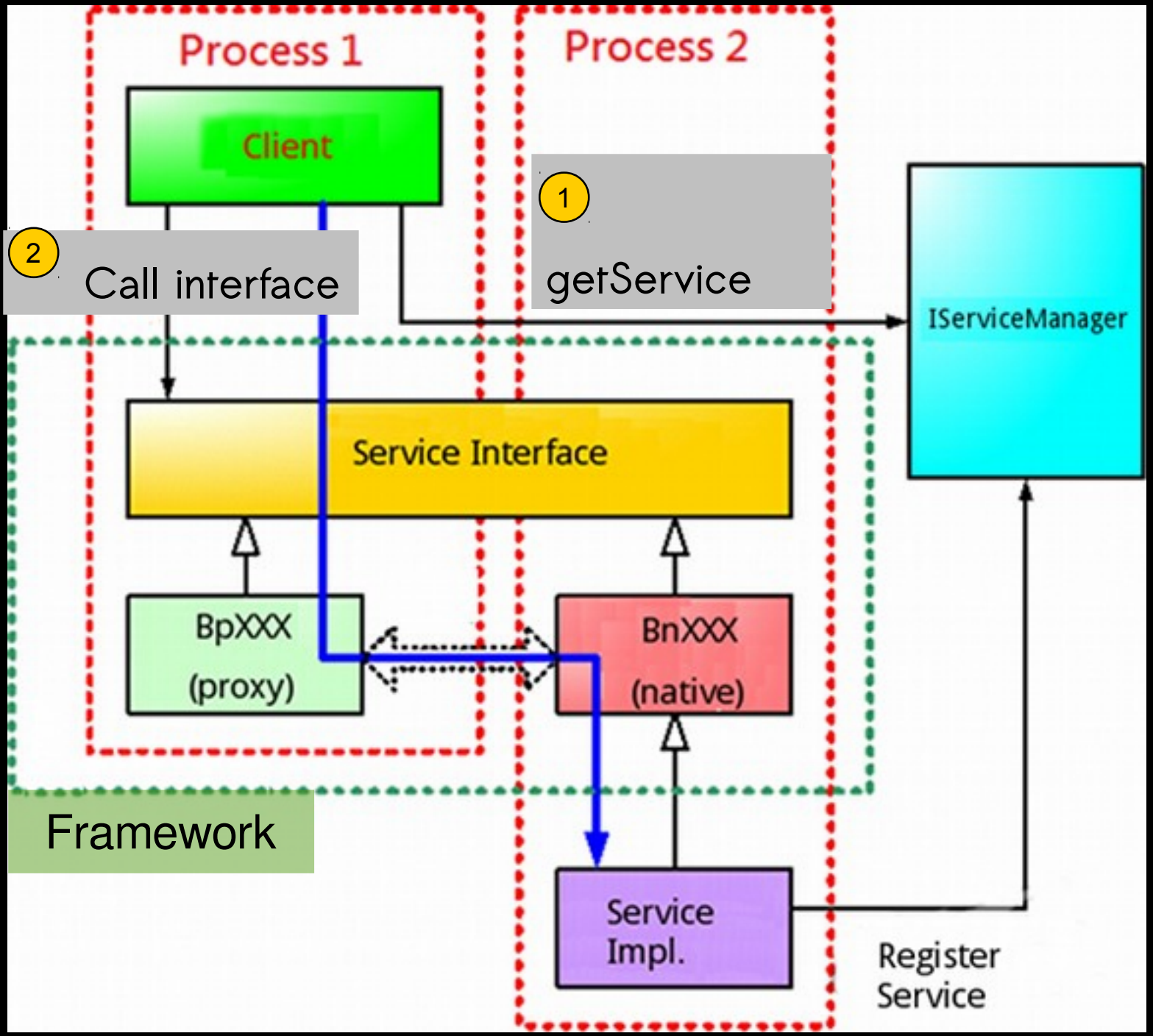


Introduce Proxy-Stub pair along with Binder


IBinder
interface
Defined by Android
Binder
class
Interface generated
by the AIDL tool
Stub
inner class
inner class
used by Android
Defined by the
application
Class that implements
the interface
AIDL
definition
Used remotely
(by the Service)
Used locally
(by clients of the Service)

IPC Interaction in Android

(Application View)

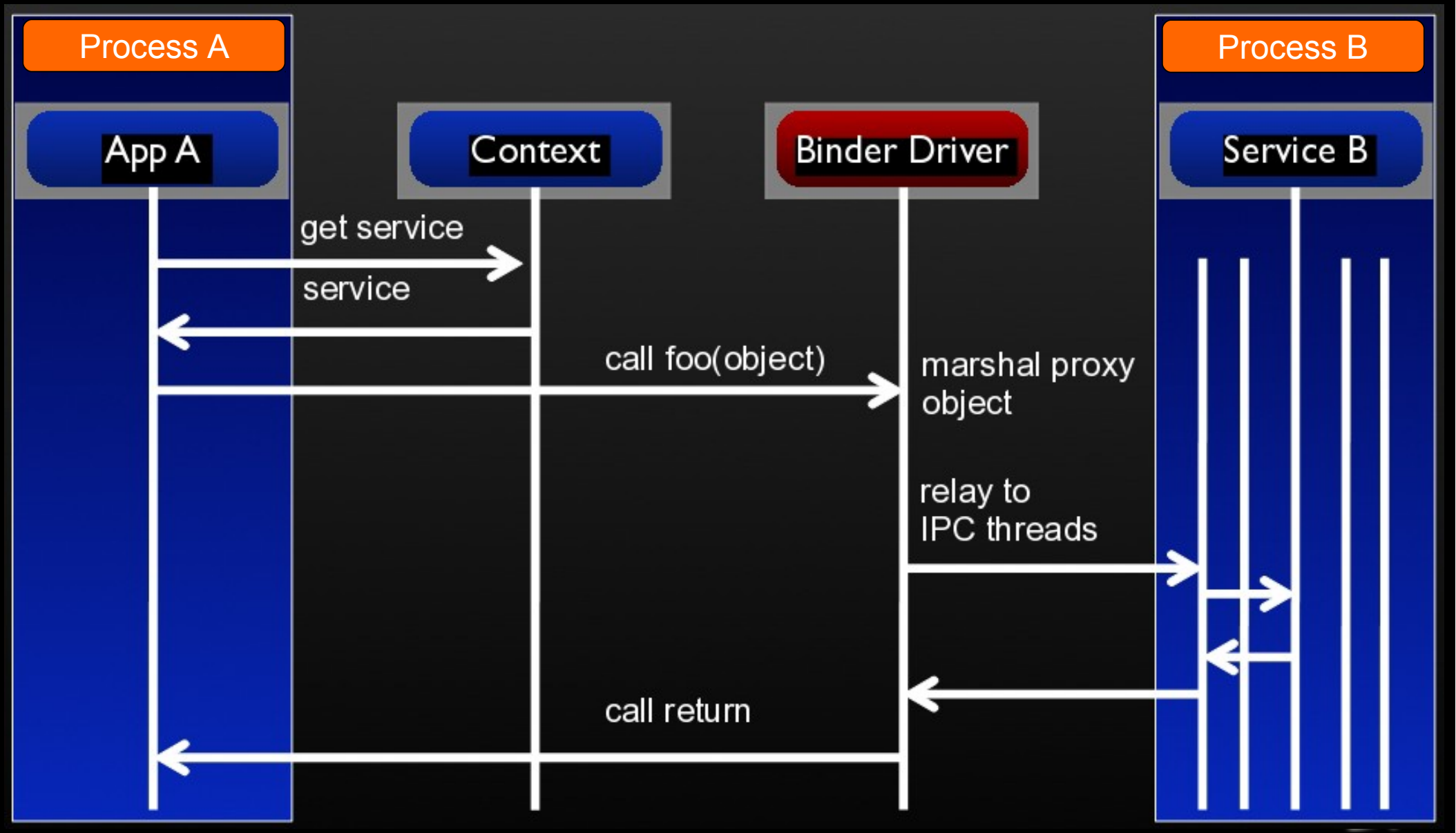


3 parts:

- BnXXX: native
- BpXXX: proxy
- Client

Invoke BpXXX

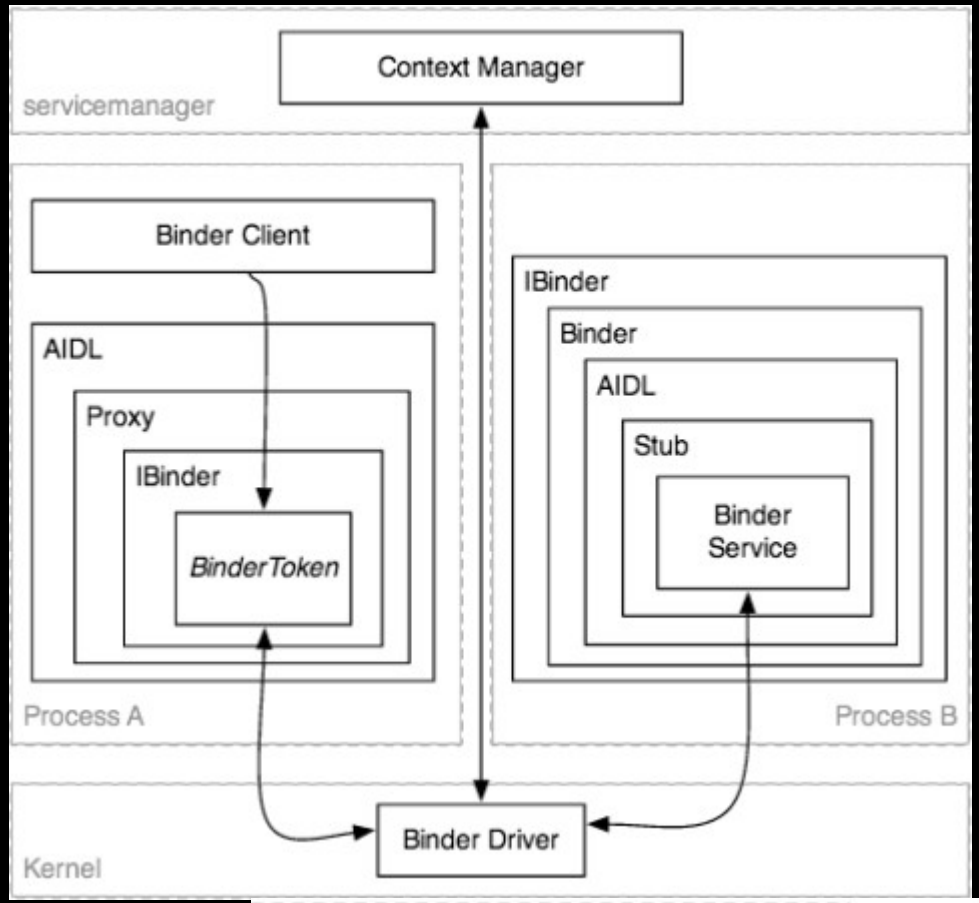
Binder in Action

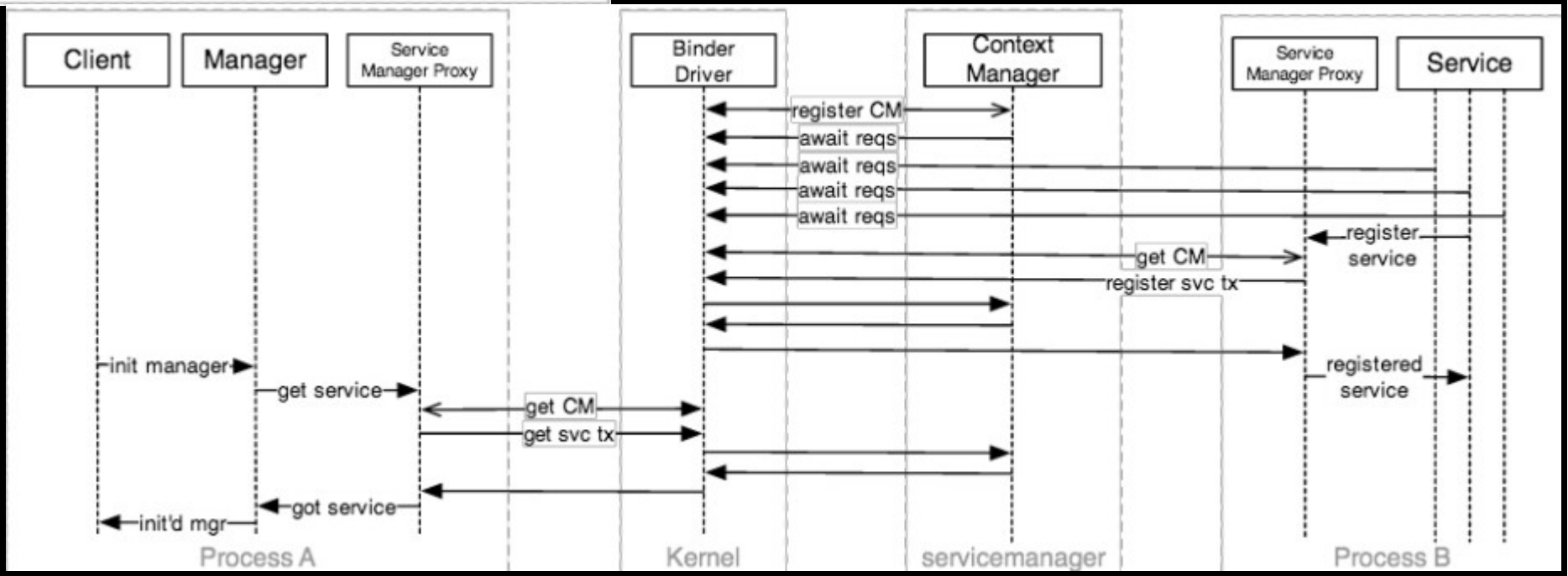


Binder Internals

Binder Terminology

- Binder
- Binder Object
  an instance of a class that implements the Binder interface.
  One Binder object can implement multiple Binders
- Binder Protocol
- IBinder Interface
  is a well-defined set of methods, properties and events that a Binder can implement.
- Binder Token
  A numeric value that uniquely identifies a Binder


Context Manager
servicemanager
Binder Client
AIDL
Proxy
IBinder
BinderToken
Process A
IBinder
Binder
AIDL
Stub
Binder Service
Process B
Binder Driver
Kernel

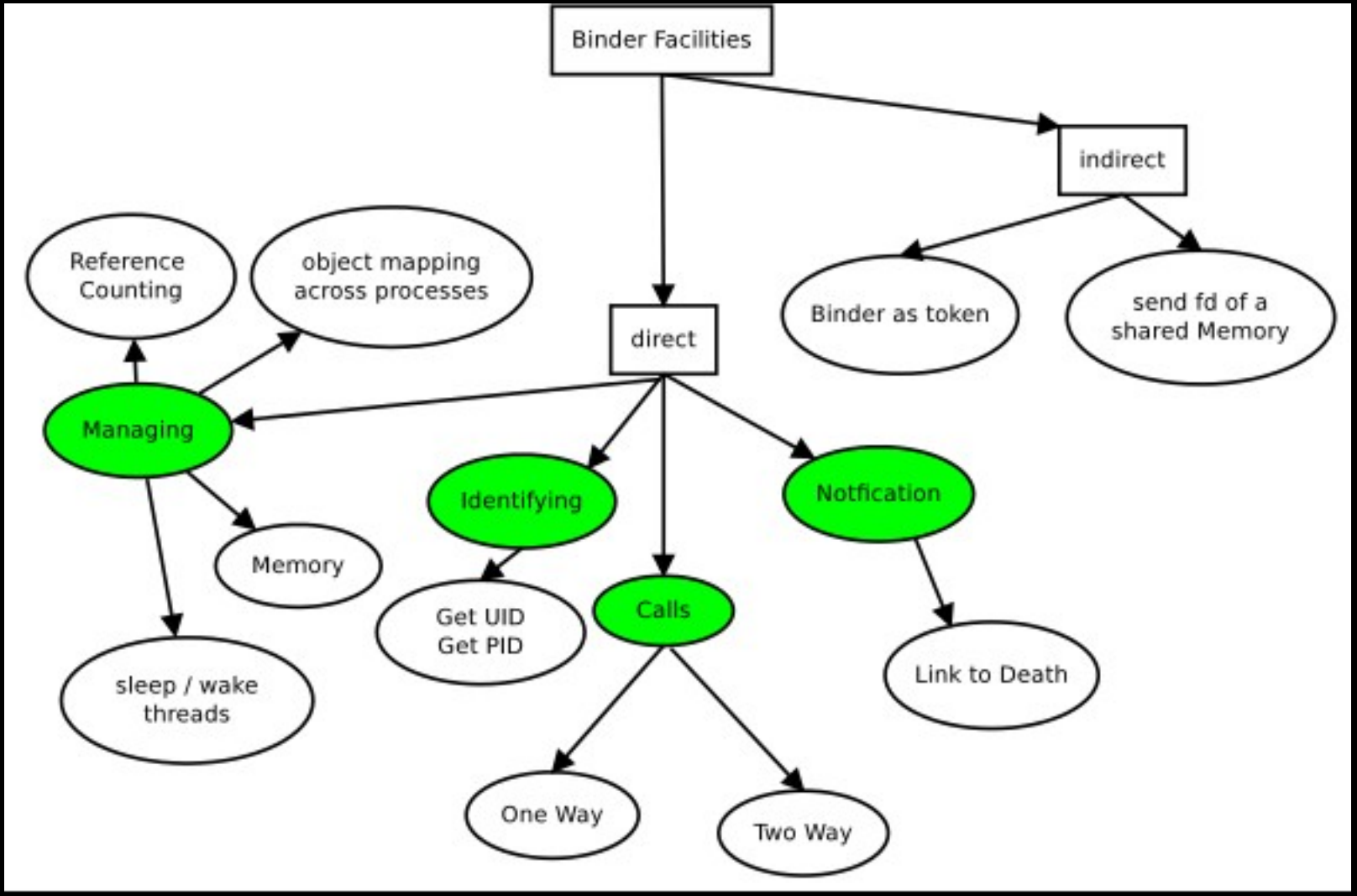

Client
Manager
Service Manager Proxy
Binder Driver
Context Manager
Service Manager Proxy
Service
register CM
await reqs
await reqs
await reqs
await reqs
register service
get CM
register svc tx
init manager
get service
registered service
get CM
get svc tx
got service
init'd mgr
Process A
Kernel
servicemanager
Process B

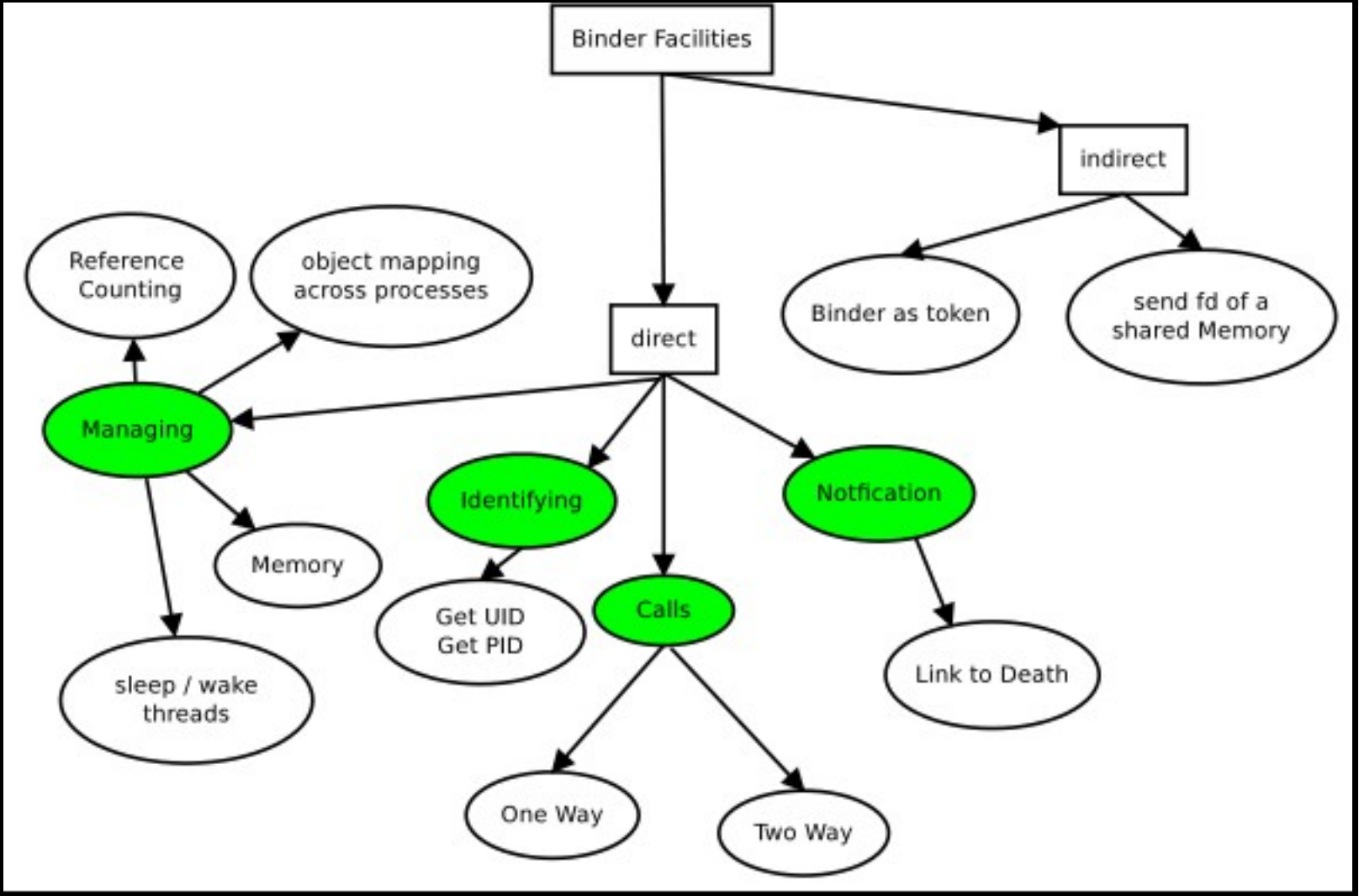
Facilities

- Simple inter process messaging system
- Managing
- Identifying
- Calls
- Notification
- Binder as a security access token

Binder simplifies the traditional RPC by abstracting its behavior.

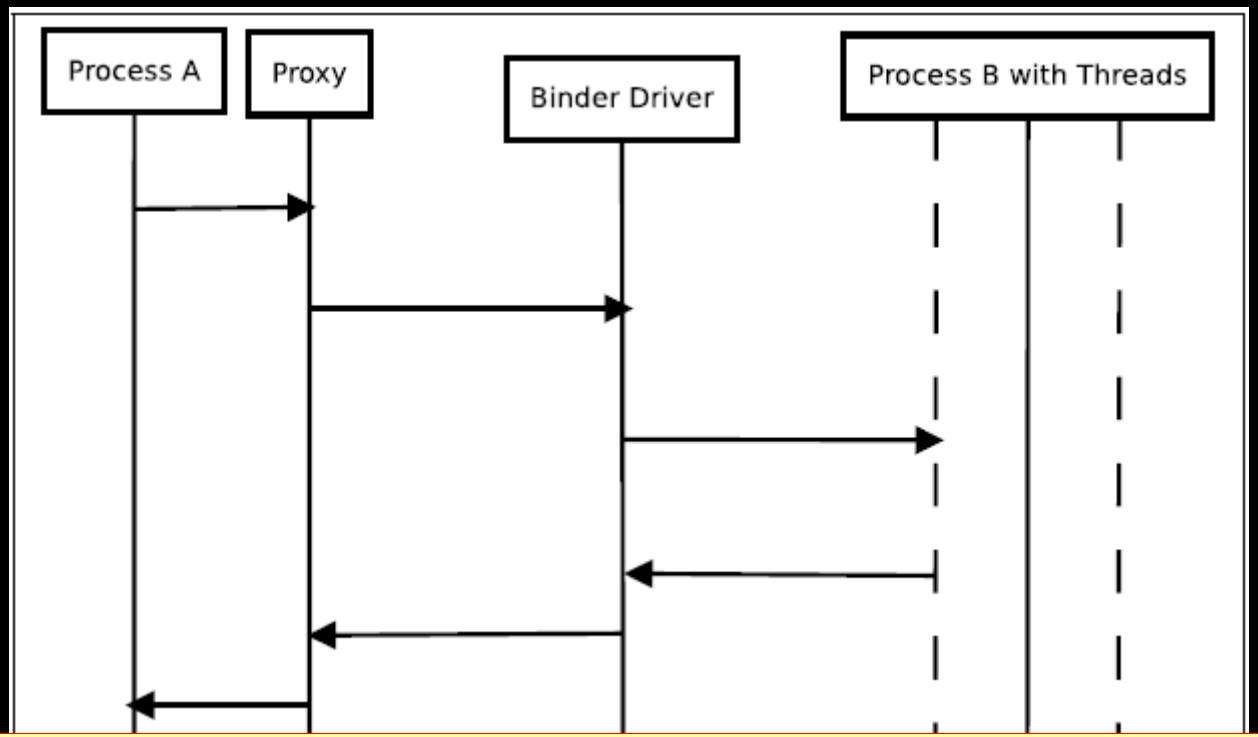
- Binder framework provides more than a simple interprocess messaging system.
- Methods on remote objects can be called as if they where local object methods.





- Facilities:
- Direct:
  - Managing
  - Identifying
  - Calls
  - Notification
- Indirect:
  - Binder as token
  - Find fd of shared memory

Communication protocol



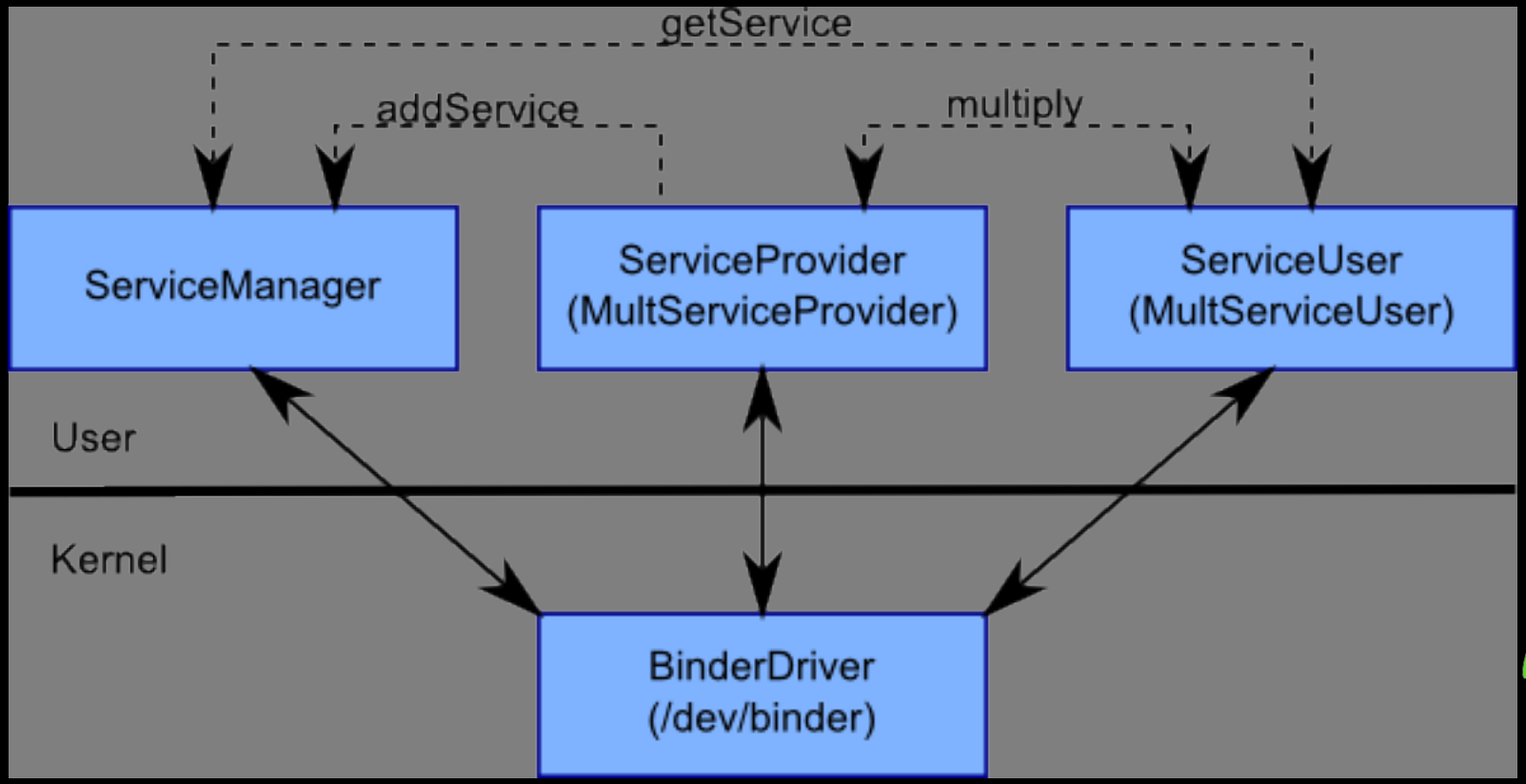
If one process sends data to another process, it is called transaction. The data is called transaction data.

Target	Binder Driver Command	Cookie	Sender ID	Data:

Target Command 0	Arguments 0
Target Command 1	Arguments 1
...	...
Target Command n-1	Arguments n-1

Service Manager (SM)

- Special Binder node with known Binder address
- Client does not know the address of remote Binder
  only Binder interface knows its own address
- Binder submits a name and its Binder token to SM
  Client retrieves Binder address with service name from SM



Get Service list from SM

```
$ adb shell service list
Found 71 services:
0    stub_isms: [com.android.internal.telephony.ISms]
1    stub_phone: [com.android.internal.telephony.ITelephony]
2    stub_iphonesubinfo:
                [com.android.internal.telephony.IPhoneSubInfo]
..
5    stub_telephony.registry:
            [com.android.internal.telephony.ITelephonyRegistry]
...
7    stub_activity: [android.app.IActivityManager]
...
9    phone: [com.android.internal.telephony.ITelephony]
…
56   activity: [android.app.IActivityManager]
...
64   SurfaceFlinger: [android.ui.ISurfaceComposer]
...
```

Call remote method in ActivityManager

```
$ adb shell service list
...
56  activity: [android.app.IActivityManager]
...
$ adb shell service call activity 1598968902
Result: Parcel(
  0x00000000: 0000001c 006e0061 00720064 0069006f '....a.n.d.r.o.i.'
  0x00000010: 002e0064 00700061 002e0070 00410049 'd...a.p.p...I.A.'
  0x00000020: 00740063 00760069 00740069 004d0079 'c.t.i.v.i.t.y.M.'
  0x00000030: 006e0061 00670061 00720065 00000000 'a.n.a.g.e.r.....')
```

```
public abstract interface IBinder {
    ...
    field public static final int INTERFACE_TRANSACTION
        = 1598968902; // 0x5f4e5446
    …
}
```

Source: frameworks/base/api/current.txt

Interact with Android Service

```
$ adb shell service call phone 1 s16 "123"
Result: Parcel(00000000 '....')
```

```
interface ITelephony {
    /* Dial a number. This doesn't place the call. It displays
     * the Dialer screen. */
    void dial(String number);
```

Source: frameworks/base/telephony/java/com/android/internal/telephony/ITelephony.aidl

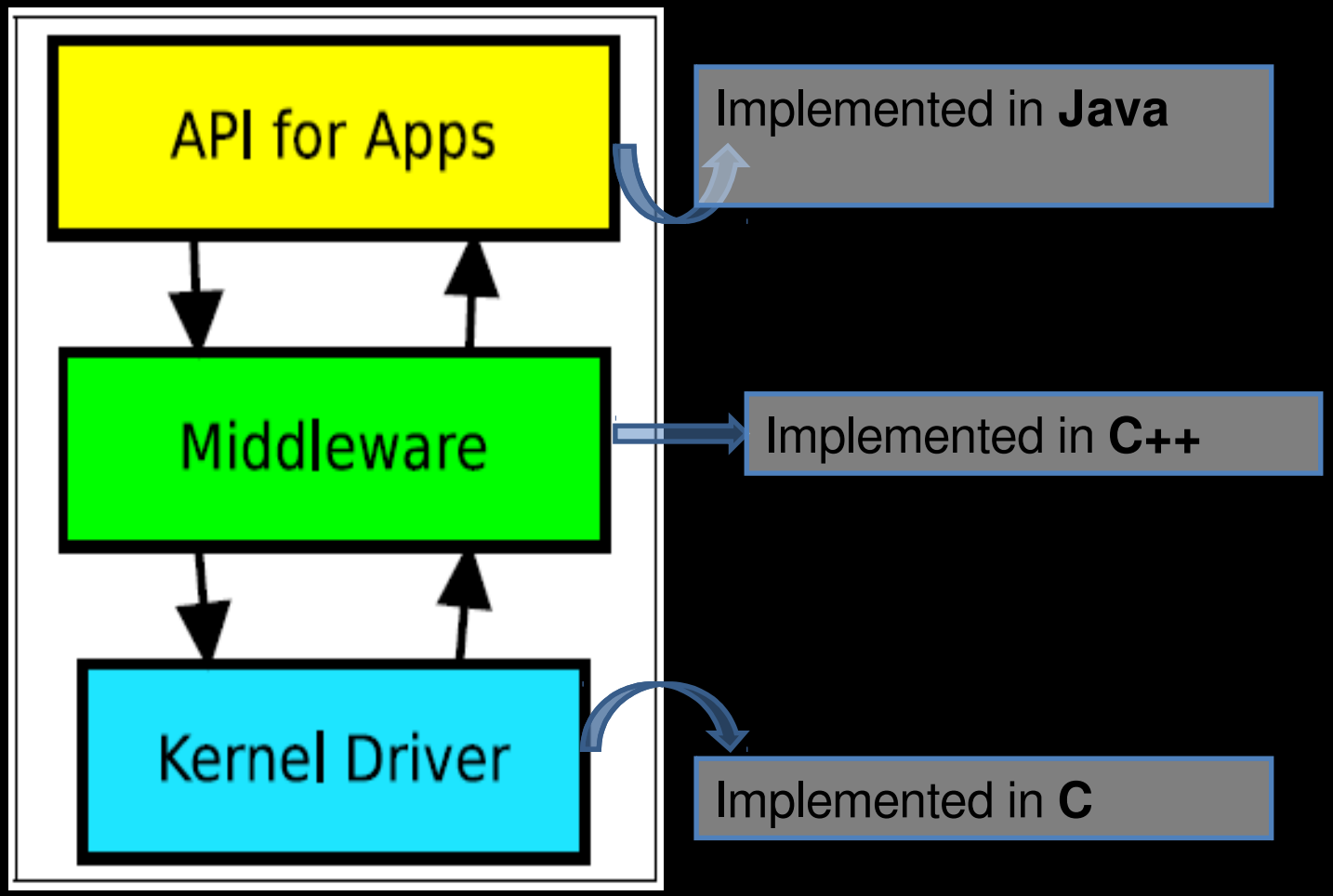
```
service call SERVICE CODE [i32 INT | s16 STR] …

Options:

   i32: Write the integer INT into the send parcel.

   s16: Write the UTF-16 string STR into the send parcel.
```

```
$ adb shell service list
Found 71 services:
...
9    phone: [com.android.internal.telephony.ITelephony]
```

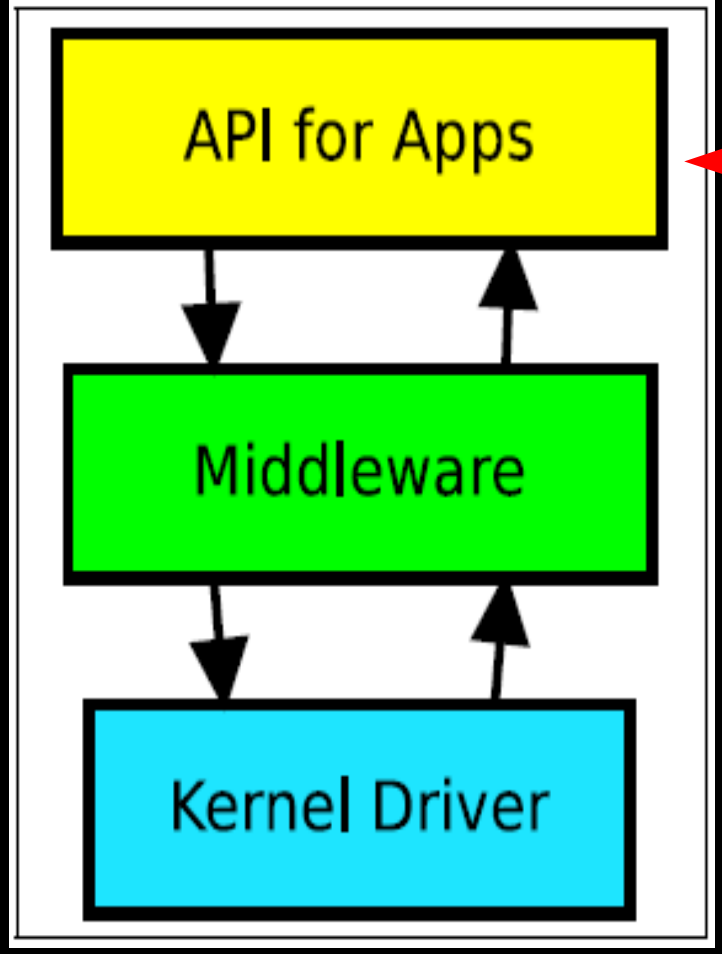
Phone Application appears in foreground.
parameter "1" → dial()
s16 "123" → String("123")

Binder and Android Framework

Implementation Layers of Binder



API Layer



- **AIDL** (Android Interface Definition Language)
  - Ease the implementation of Android remote services
  - Defines an interface with method of remote services
  - AIDL parser generates Java class
    - Proxy class for Client
    - Stub class for Service
- **Java API Wrapper**
  - Introduce facilities to the binder
  - Wraps the middleware layer

AIDL

- Data Types
  - Java Primitives
  - Containers
    - String, List, Map, CharSequence
    - List<>
    - Multidimensional Array
  - Parcelable
  - Interface Reference
- Direction: in, out, inout
- oneway
  - android.os.IBinder.FLAG_ONEWAY

AIDL Compiler

- Full-fledged Java(-only) Support
- Stub and Proxy Generator

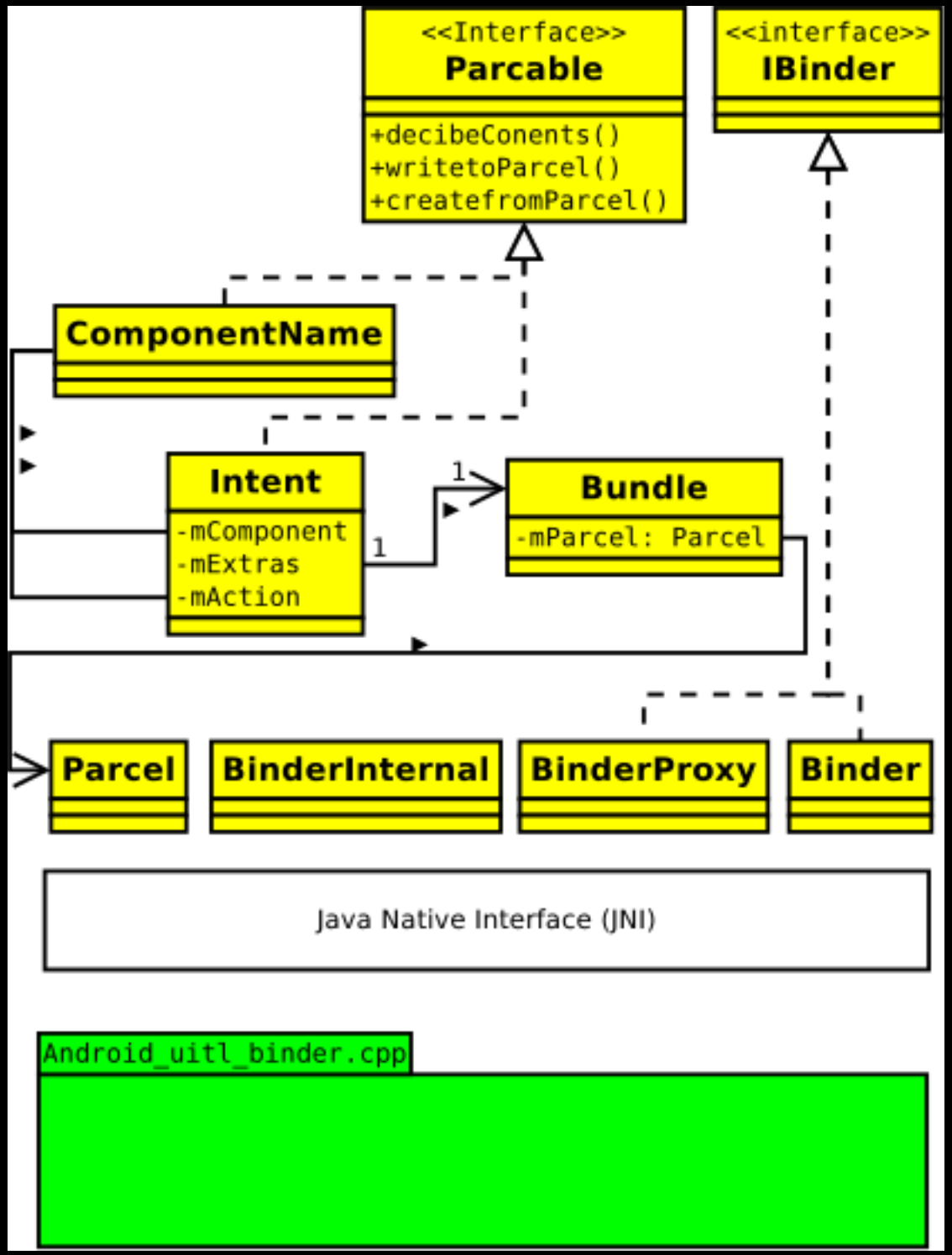
```
// Interface
interface IRemoteService {
    void ping();
}
```

Client

```
IRemoteService mService =
    IRemoteService.Stub.asInterface(service);
```

```
public class RemoteService extends Service {
    public IBinder onBind(Intent intent) { return mBinder; }
    private final IRemoteService.Stub mBinder =
        new IRemoteService.Stub() {
            public void ping() { // Nothing }
    };
}
```

Server


<<Interface>>
Parcable
+decibeConents()
+writetoParcel()
+createfromParcel()
<<interface>>
IBinder
ComponentName
Intent
-mComponent
-mExtras
-mAction
1
1
Bundle
-mParcel: Parcel
Parcel
BinderInternal
BinderProxy
Binder
Java Native Interface (JNI)
Android_uitl_binder.cpp

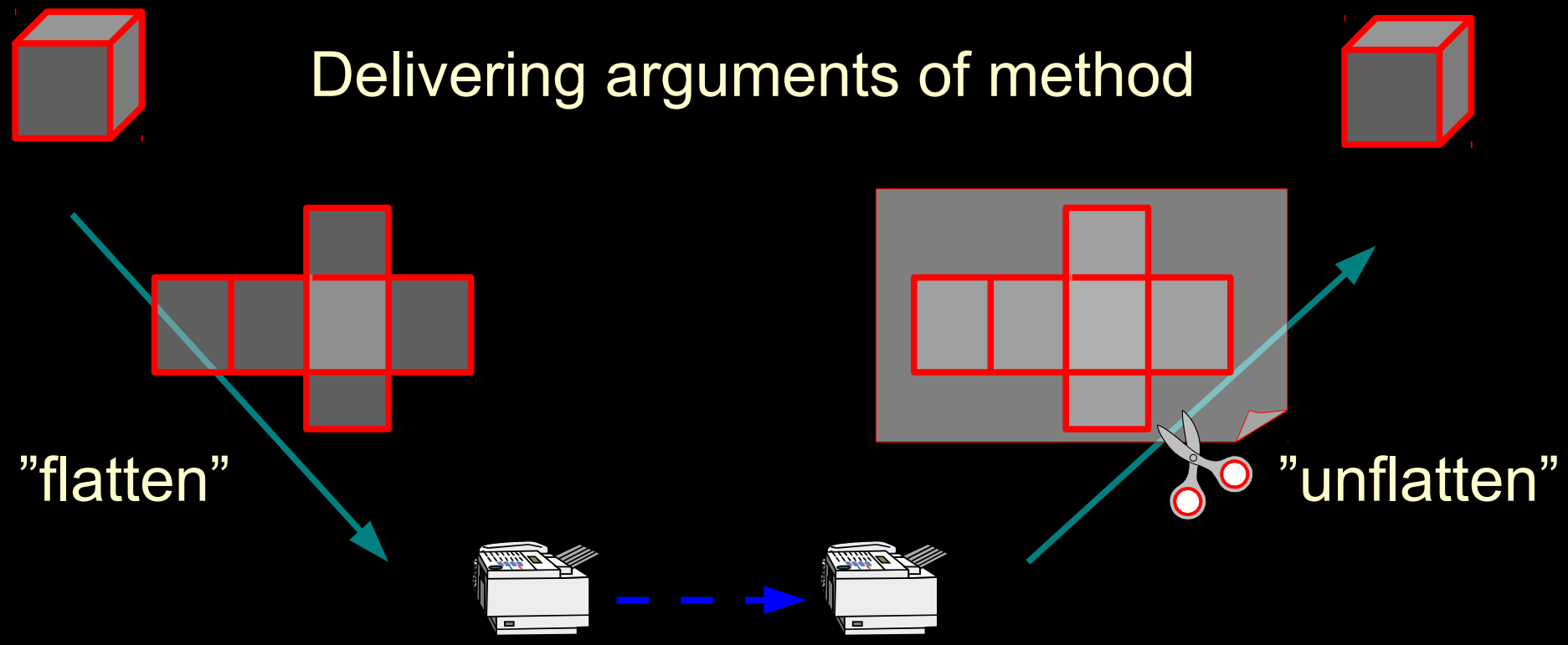
Parcels and Marshalling

- Simple inter process messaging system
- In an object oriented view, the transaction data is called parcel.
- The procedure of building a parcel is called **marshalling** an object.
- The procedure of rebuilding a object from a parcel is called **unmarshalling** an object.

Parcel

- Marshalling – The transferring of data across process boundaries
  - Represented in native binary encoding
- Mostly handled by AIDL-generated code
- Extensible – Parcelable

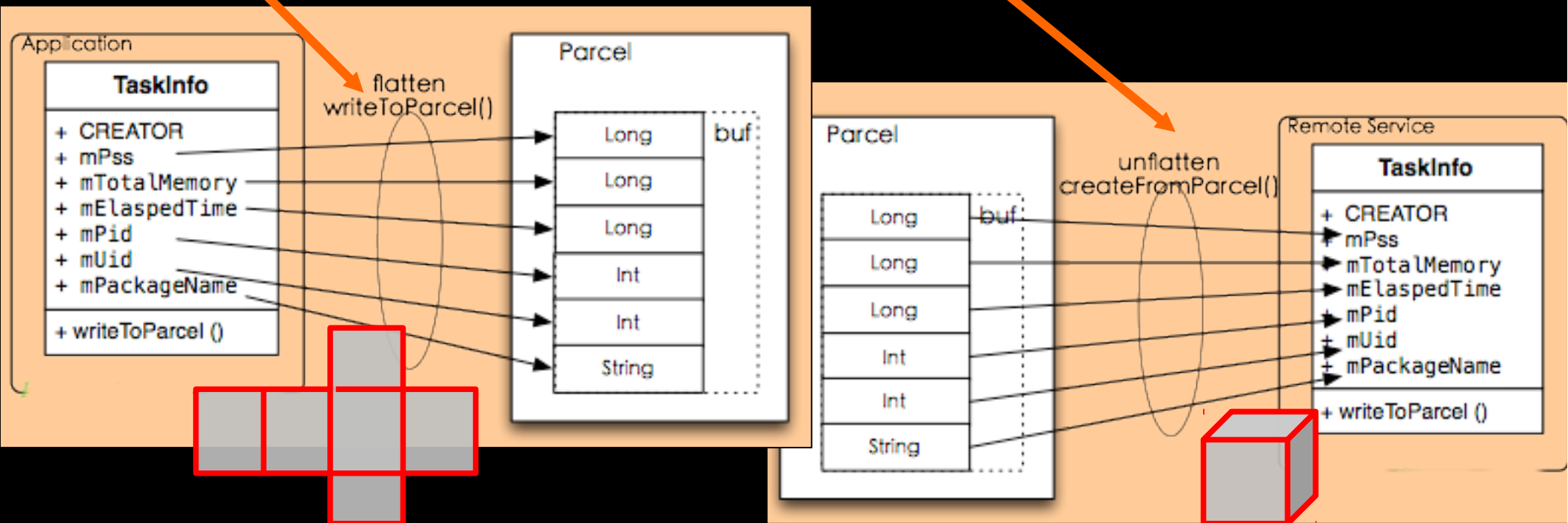
android.os.Parcel

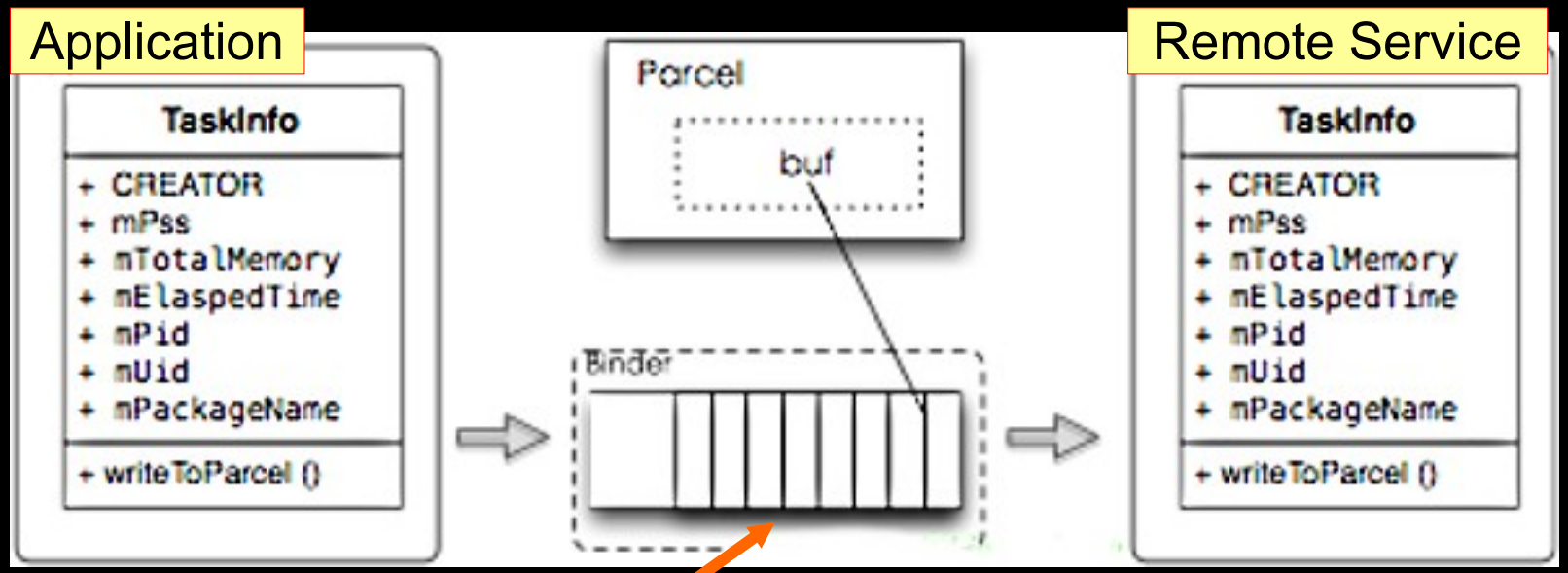


Source: Inter-process communication of Android, Tetsuyuki Kobayashi

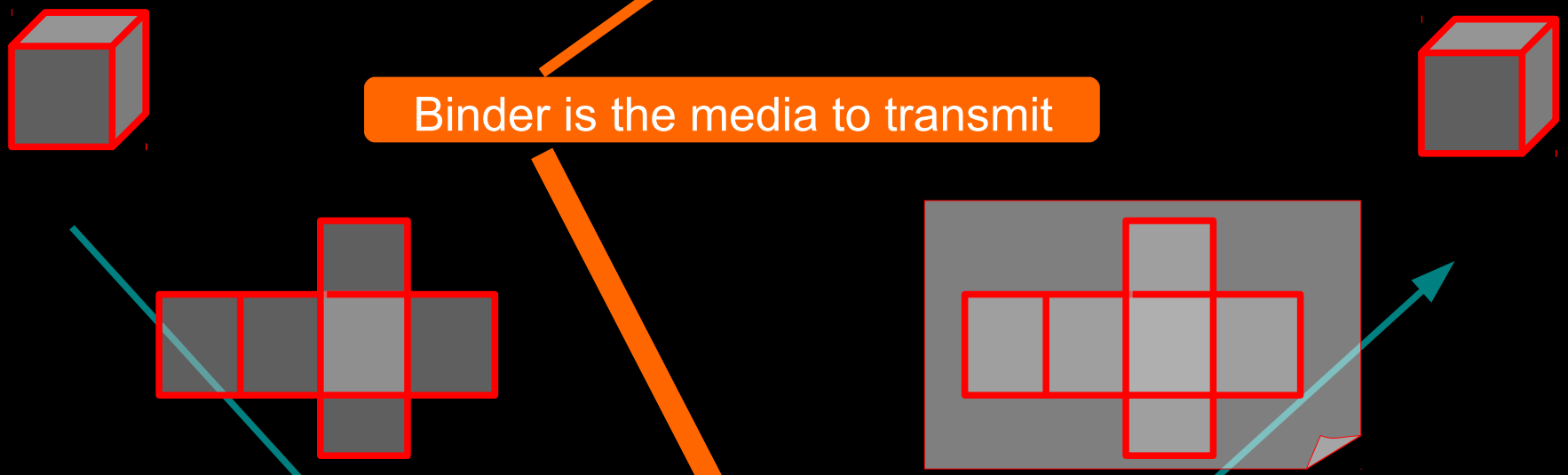
```
public class TaskInfo implements android.os.Parcelable {
    public long mPss, mTotalMemory, mElapsedTime;
    public int mPid, mUid;
    public String mPackageName;
    TaskInfo() { ... }
    public void writeToParcel(Parcel out, int flags) { … }
    public static final Parcelable.Creator<TaskInfo>CREATOR =
        new Parcelable.Creator<TaskInfo>() { … }
    public TaskInfo createFromParcel(Parcel in)  {
        return TaskInfo(in); }
    private TaskInfo(Parcel in) { … }
```

class TypeInfo as example



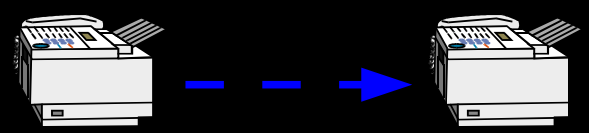


Binder is the media to transmit



”flatten”

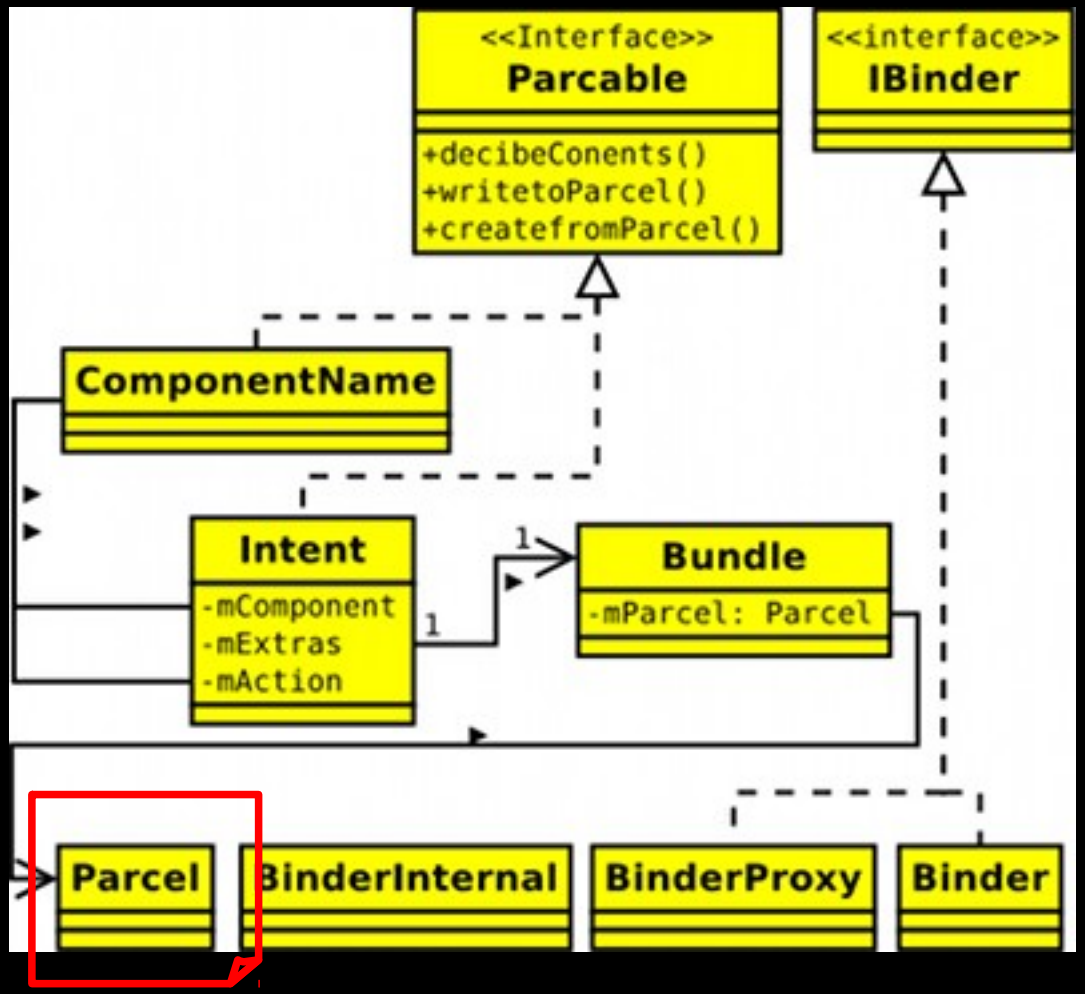
”unflatten”



Transmit object as FAX

Parcel Definition

- Container for a message (data and object references) that can be sent through an IBinder.



- A Parcel can contain both flattened data that will be unflattened on the other side of the IPC (using the various methods here for writing specific types, or the general Parcelable interface), and references to live IBinder objects that will result in the other side receiving a proxy IBinder connected with the original IBinder in the Parcel.

Representation of Parcel

- Parcel is not for general-purpose serialization
  This class (and the corresponding Parcelable API for placing arbitrary objects into a Parcel) is designed as a high-performance IPC transport.
  Not appropriate to place any Parcel data into persistent storage
- Functions for writing/reading primitive data types:

```
writeByte(byte) / readByte()
writeDouble(double) / readDouble()
writeFloat(float) / readFloat()
writeInt(int) / readInt()
writeLong(long) / readLong()
writeString(String) / readString()
```

Parcelable

- The Parcelable protocol provides an extremely efficient (but low-level) protocol for objects to write and read themselves from Parcels.
- Use the direct methods to write/read

```
writeParcelable(Parcelable, int)
readParcelable(ClassLoader)
writeParcelableArray(T[],int)
readParcelableArray(ClassLoader)
```

- These methods write both the class type and its data to the Parcel, allowing that class to be reconstructed from the appropriate class loader when later reading.

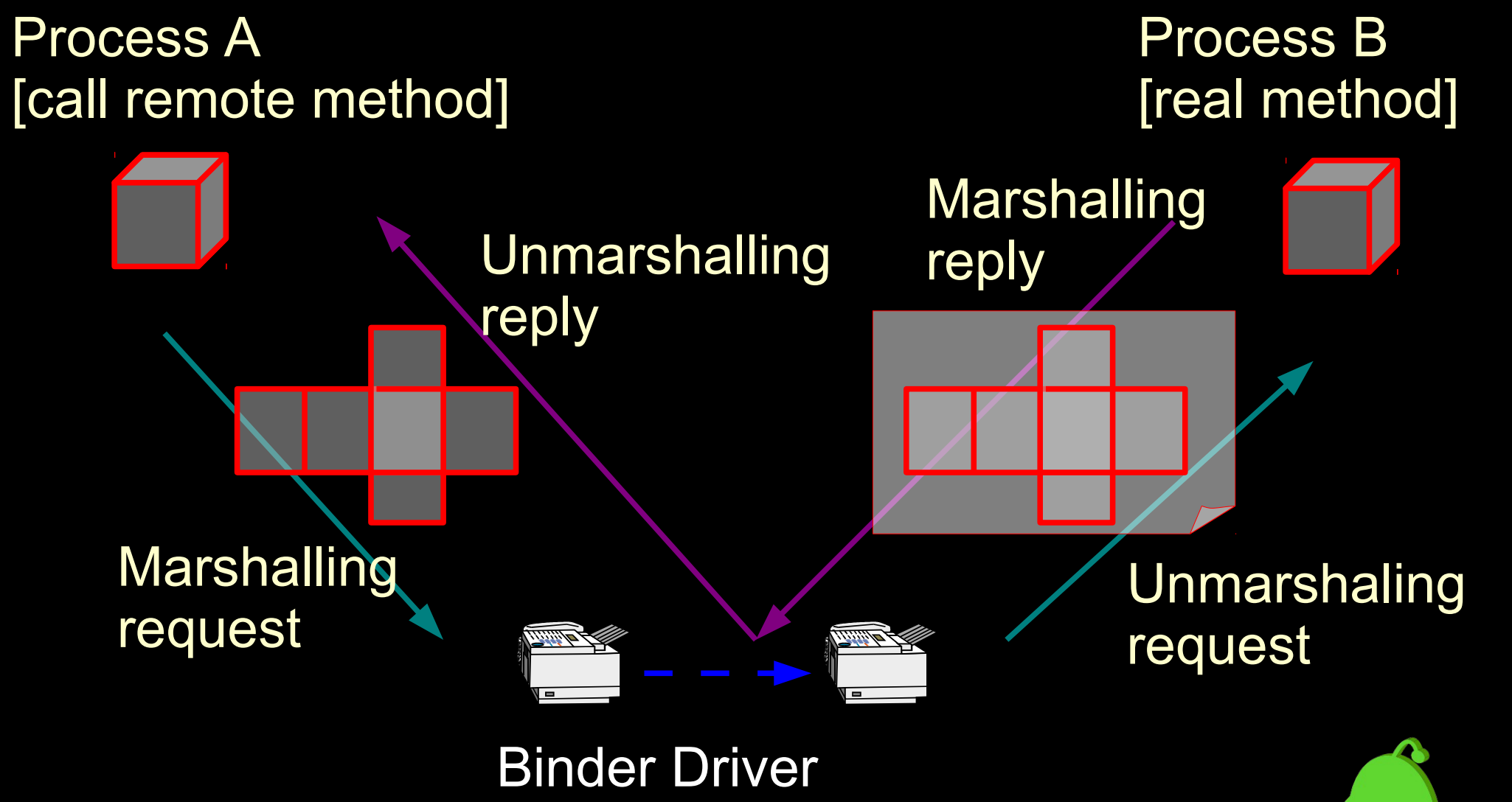
Parcelable

- Implement the Parcelable interface.
  implement `writeToParcel()` and `readFromParcel()`.
  Note: the order in which you write properties must be the same as the order in which you read them.
- Add a static final property to the class with the name CREATOR .
  The property needs to implement the **`android.os.Parcelable.Creator<T>`** interface.
- Provide a constructor for the Parcelable that knows how to create the object from the Parcel.
- Define a Parcelable class in an .aidl file that matches the .java file containing the complex type .
  AIDL compiler will look for this file when compiling your AIDL files.

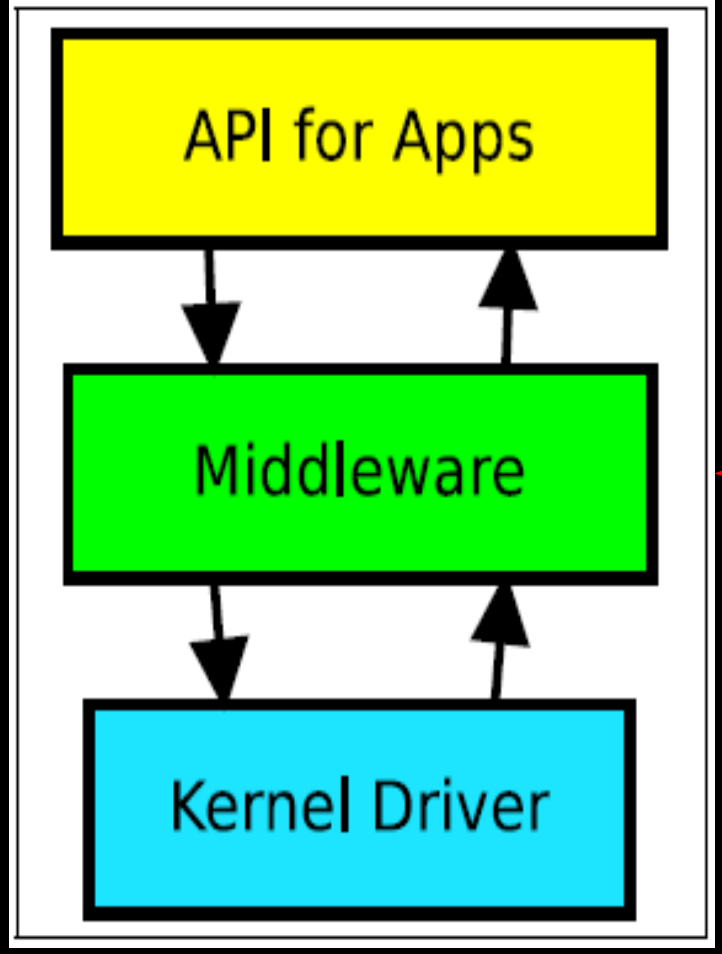
Bundles

- A special type-safe container, called Bundle, is available for key/value maps of heterogeneous values.
- This has many optimizations for improved performance when reading and writing data, and its type-safe API avoids difficult to debug type errors when finally marshalling the data contents into a Parcel.

RPC Implementation in Binder

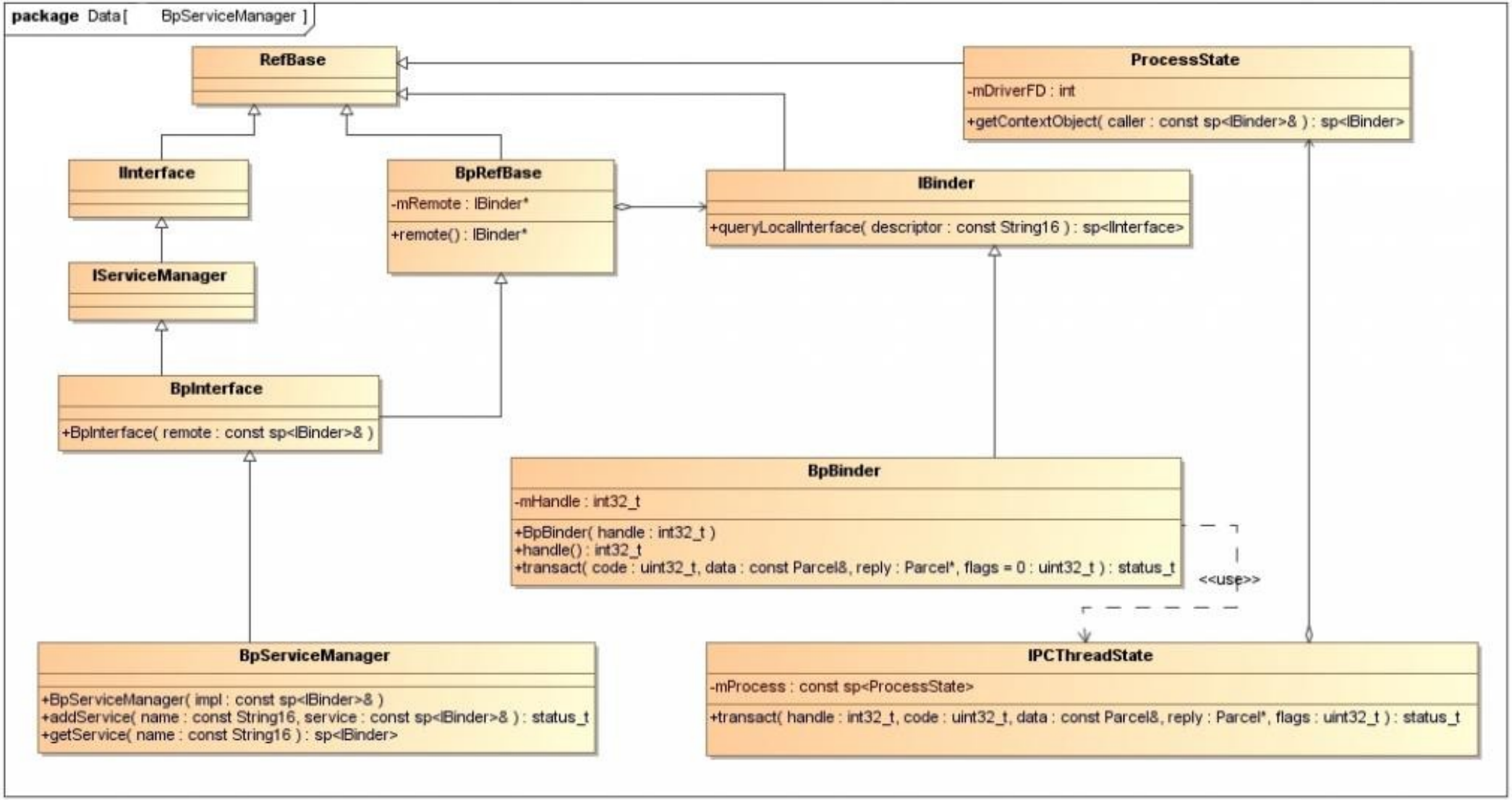


Middleware Layer



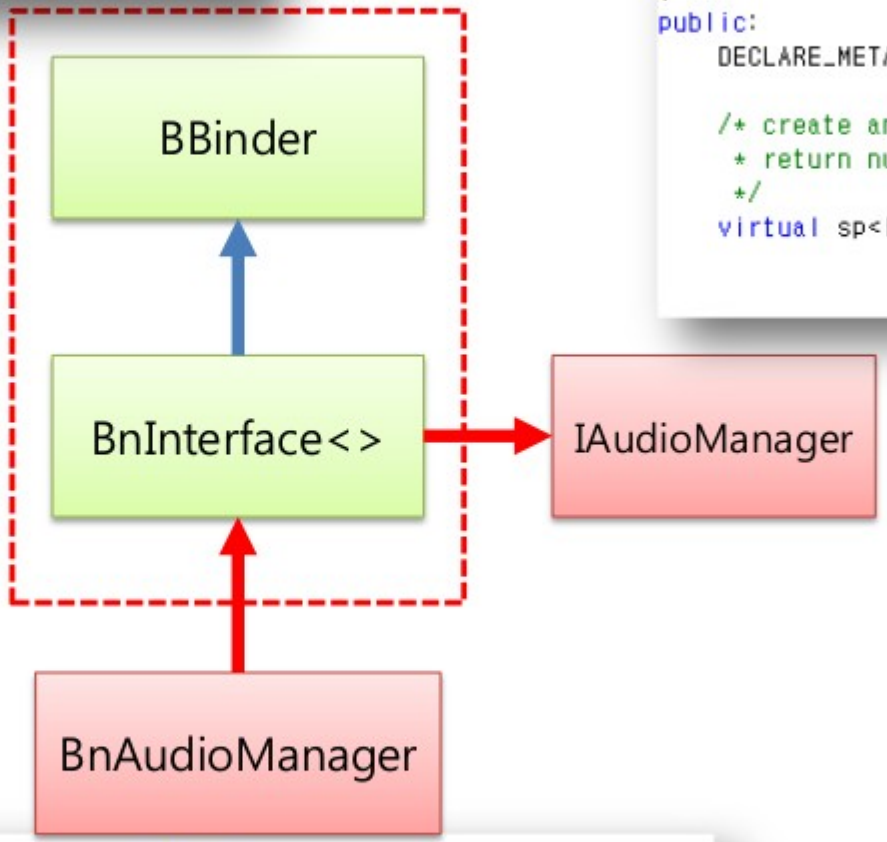
- Implements the user space facilities of the Binder framework in C++
- Implements structures and methods to spawn and manage new threads
- Marshalling and unmarshalling of specific data
- Provides interaction with the Binder kernel driver

- frameworks/base/include/binder/IServiceManager.h
  sp<IServiceManager> defaultServiceManager()
- frameworks/base/include/binder/IInterface.h
  template **BpInterface**



```
template<typename INTERFACE>
class BnInterface : public INTERFACE, public BBinder
{
public:
    virtual sp<IInterface>      queryLocalInterface(const String16
    virtual const String16&     getInterfaceDescriptor() const;

protected:
    virtual IBinder*            onAsBinder();
};
```

```
class IAudioFlinger : public IInterface
{
public:
    DECLARE_META_INTERFACE(AudioFlinger);

    /* create an audio track and registers
     * return null if the track cannot be
     */
    virtual sp<IAudioTrack> createTrack(
                                pid_t pid,
                                int stream
```




```
class BnAudioFlinger : public BnInterface<IAudioFlinger>
{
public:
    virtual status_t    onTransact( uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);
};
```

BpServiceManager
addService()

```
class BpServiceManager : public BpInterface<IServiceManager>
{
public:
    BpServiceManager(const sp<IBinder>& impl)
        : BpInterface<IServiceManager>(impl)
    {
    }

    virtual sp<IBinder> getService(const String16& name) const { ... }

    virtual sp<IBinder> checkService( const String16& name) const { ... }

    virtual status_t addService(const String16& name, const sp<IBinder>& service)
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        data.writeString16(name);
        data.writeStrongBinder(service);
        status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
        return err == NO_ERROR ? reply.readExceptionCode() : err;
    }

    virtual Vector<String16> listServices() { ... }
};
```

BpBinder
transact()

```
class BnServiceManager : public BnInterface<IServiceManager>
{
public:
    virtual status_t    onTransact( uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);
};
```

BnServiceManager
onTransact()

```
status_t BnServiceManager::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    //printf("ServiceManager received: "); data.print();
    switch(code) {
...
        case ADD_SERVICE_TRANSACTION: {
            CHECK_INTERFACE(IServiceManager, data, reply);
            String16 which = data.readStrong16();
            sp<IBinder> b = data.readStrongBinder();
            status_t err = addService(which, b);
            reply->writeInt32(err);
            return NO_ERROR;
        } break;
...
        default:
            return BBinder::onTransact(code, data, reply, flags);
    }
}
```

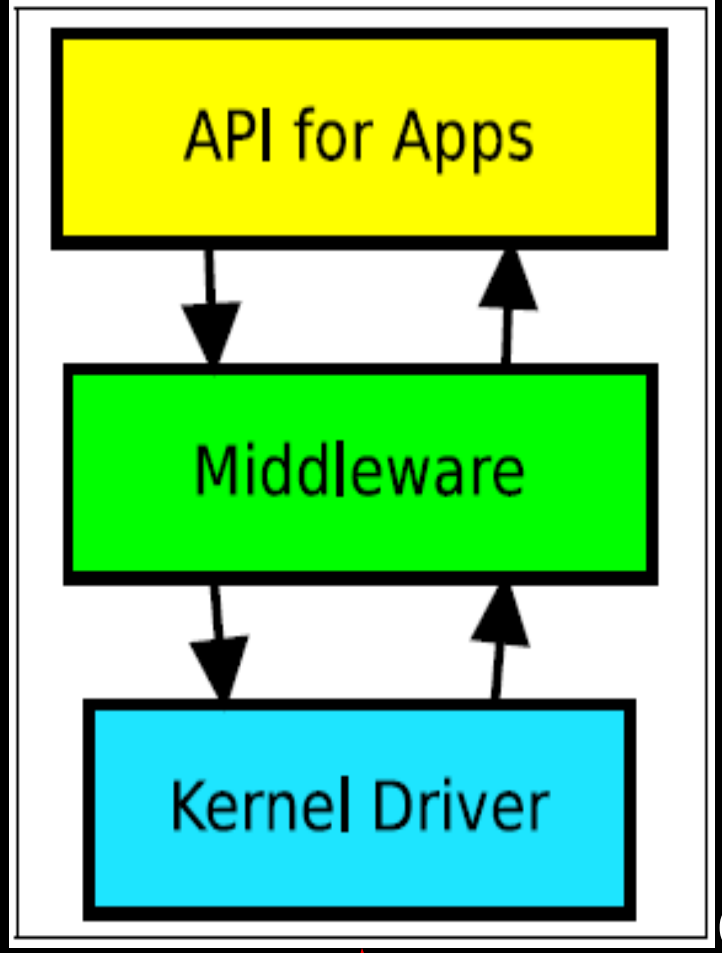
ServiceManager
addService()

IPCThreadState

- Each Binder runtime is represented by an instance of class ProcessState.
- This instance is created by the first call to ProcessState.self
- Flow:

  IPCThreadState.executeCommand → BBinder.transact → JavaBBinder.onTransact → Binder.execTransact → Binder.onTransact

Kernel Driver Layer

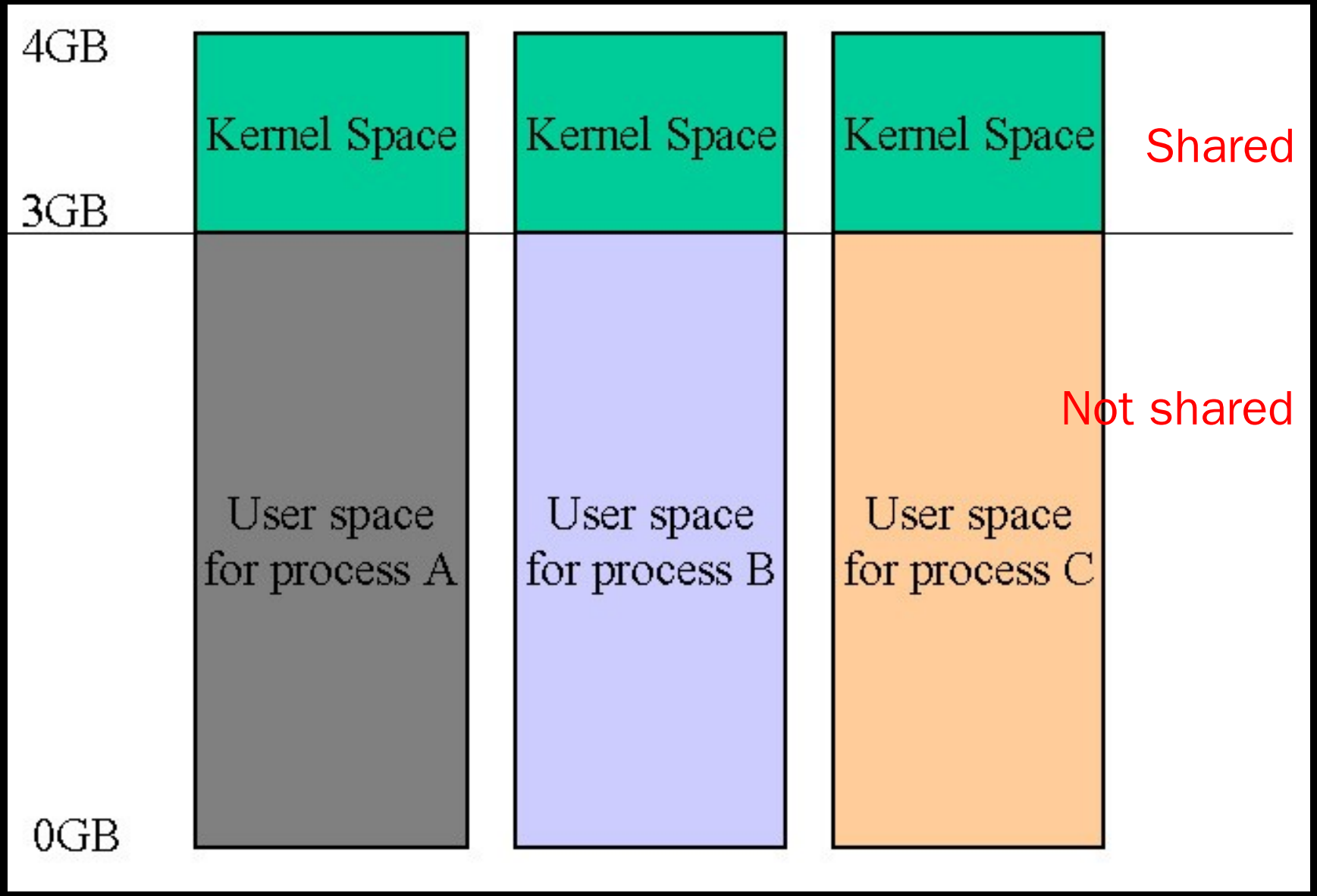


- Binder Driver supports the file operations open, mmap, release, poll and the system call ioctl
- ioctl arguments
  - Binder driver command code
  - Data buffer

Command codes

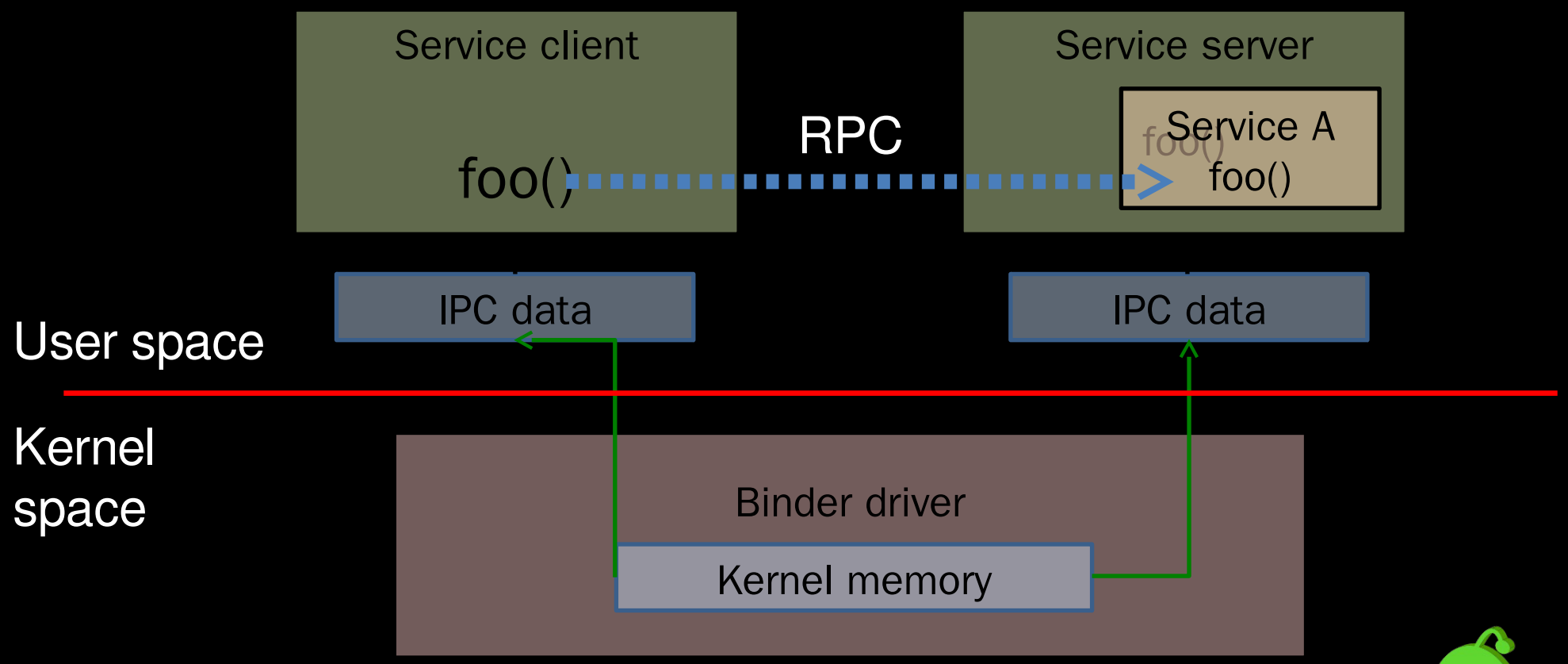
- BINDER_WRITE_READ
- BINDER_SET_MAX_THREADS
- BINDER_SET_CONTEXT_MGR
- BINDER_THREAD_EXIT
- BINDER_VERSION

Process Address Space

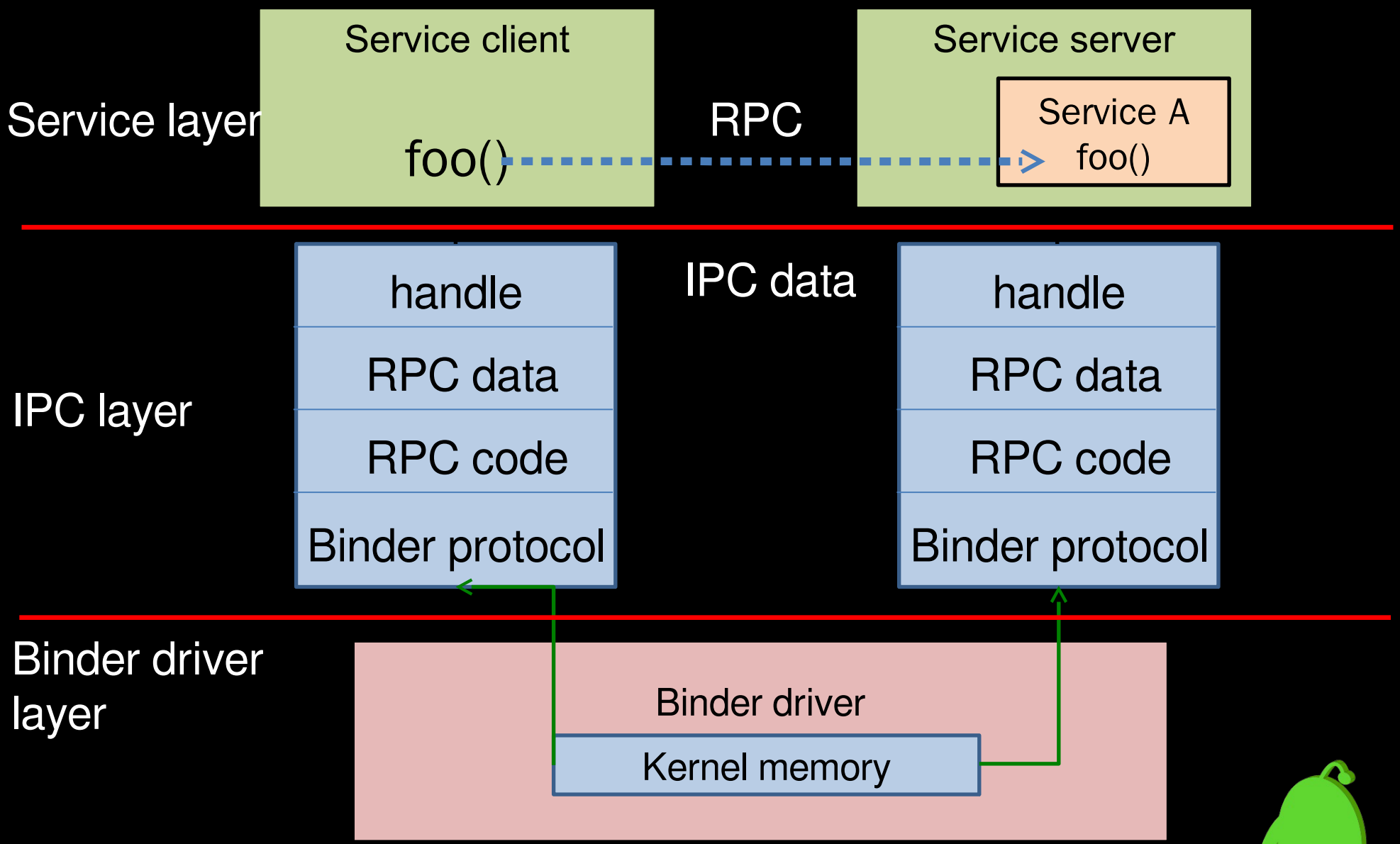


Binder Driver

- Implements remote procedure call (RPC) mechanism

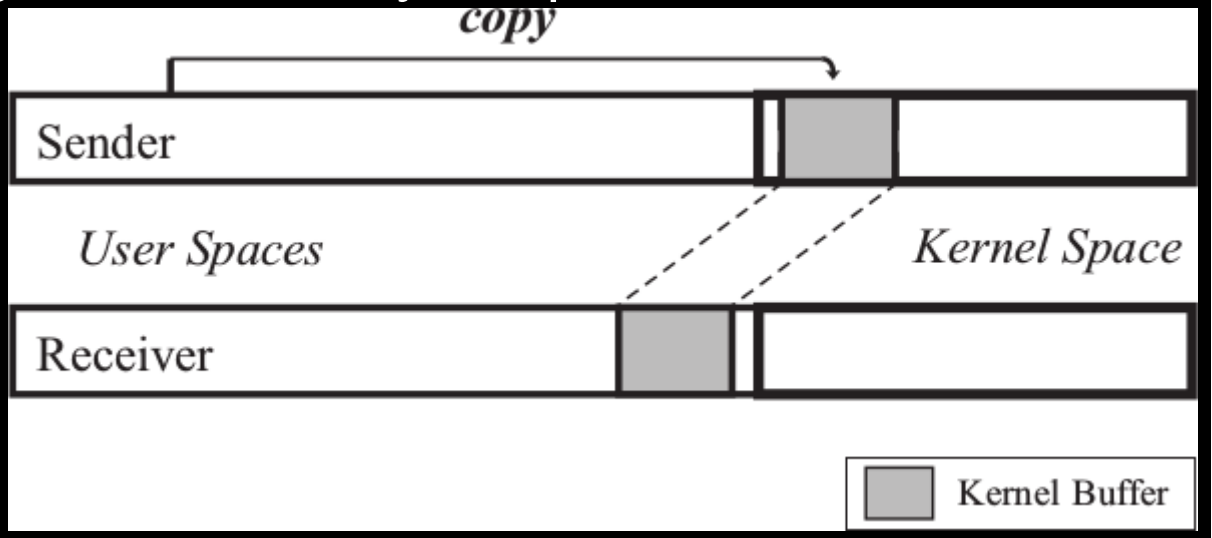


Flow of Binder IPC Data



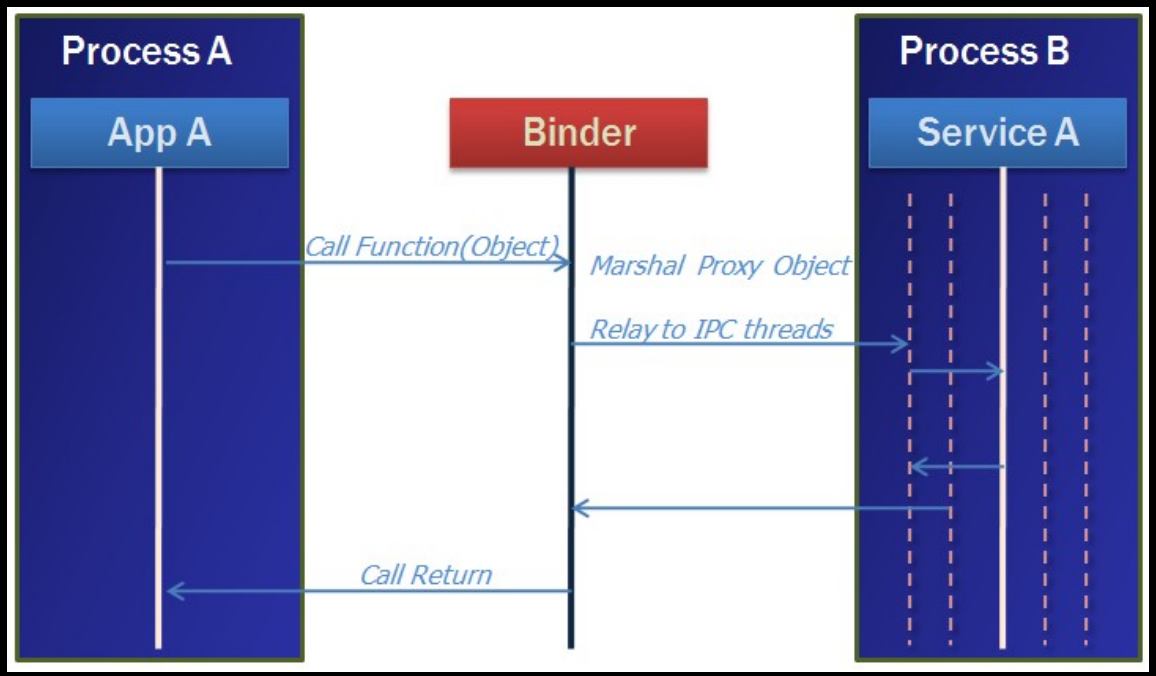
Efficient IPC

- Adopts a direct-message-copy scheme and requires only single data copy to transfer the IPC payload.
  - → To receive data from Binder driver, a process first calls mmap() on the Binder driver's device file and allows Binder driver to manage a part of its memory address space. Binder driver then allocates a memory buffer in the kernel space, called kernel buffer, and maps the kernel buffer to the process's address space.
  - → After that, when any process sends data to this process, the Binder driver only needs to copy the data from the sender process's memory space to the kernel buffer.
  - → The data will then be available in the receiver process's memory space.
- As opposed to the mechanisms that require two-fold data copy from sender process to kernel and kernel to receiver process, such as pipes or sockets, the Binder driver only requires a single data copy and therefore has lower latency, CPU usage, and memory footprint.

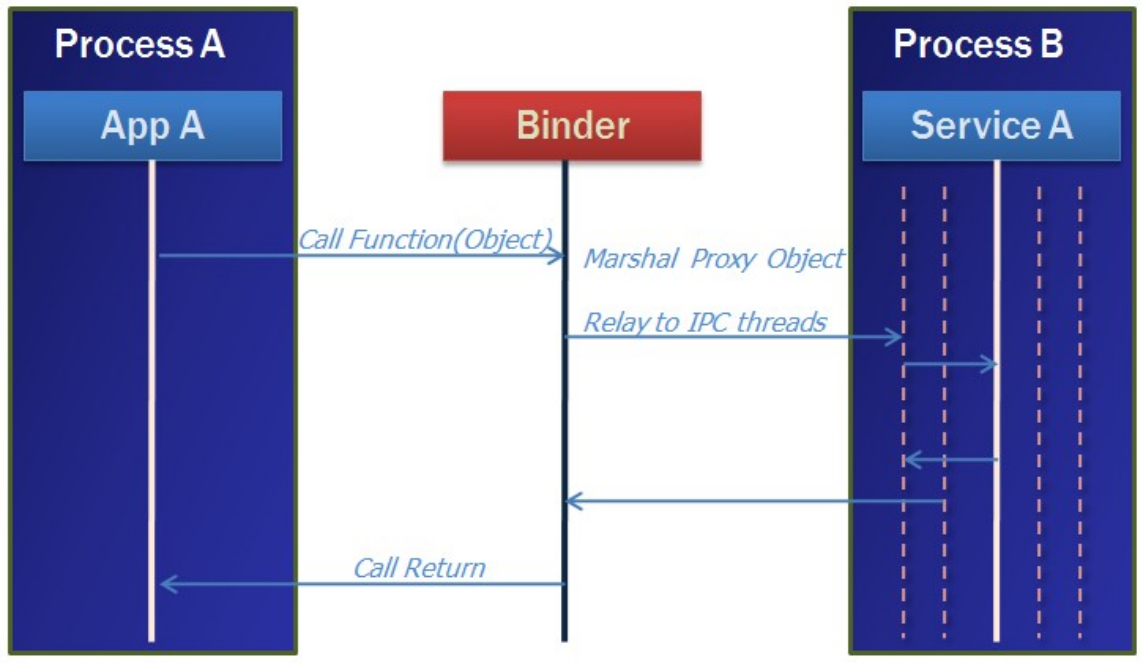


Binder Driver

- Multi-thread aware
  - Have internal status per thead
  - Compare to UNIX socket: sockets have internal status per file descriptor (FD)



Binder Driver

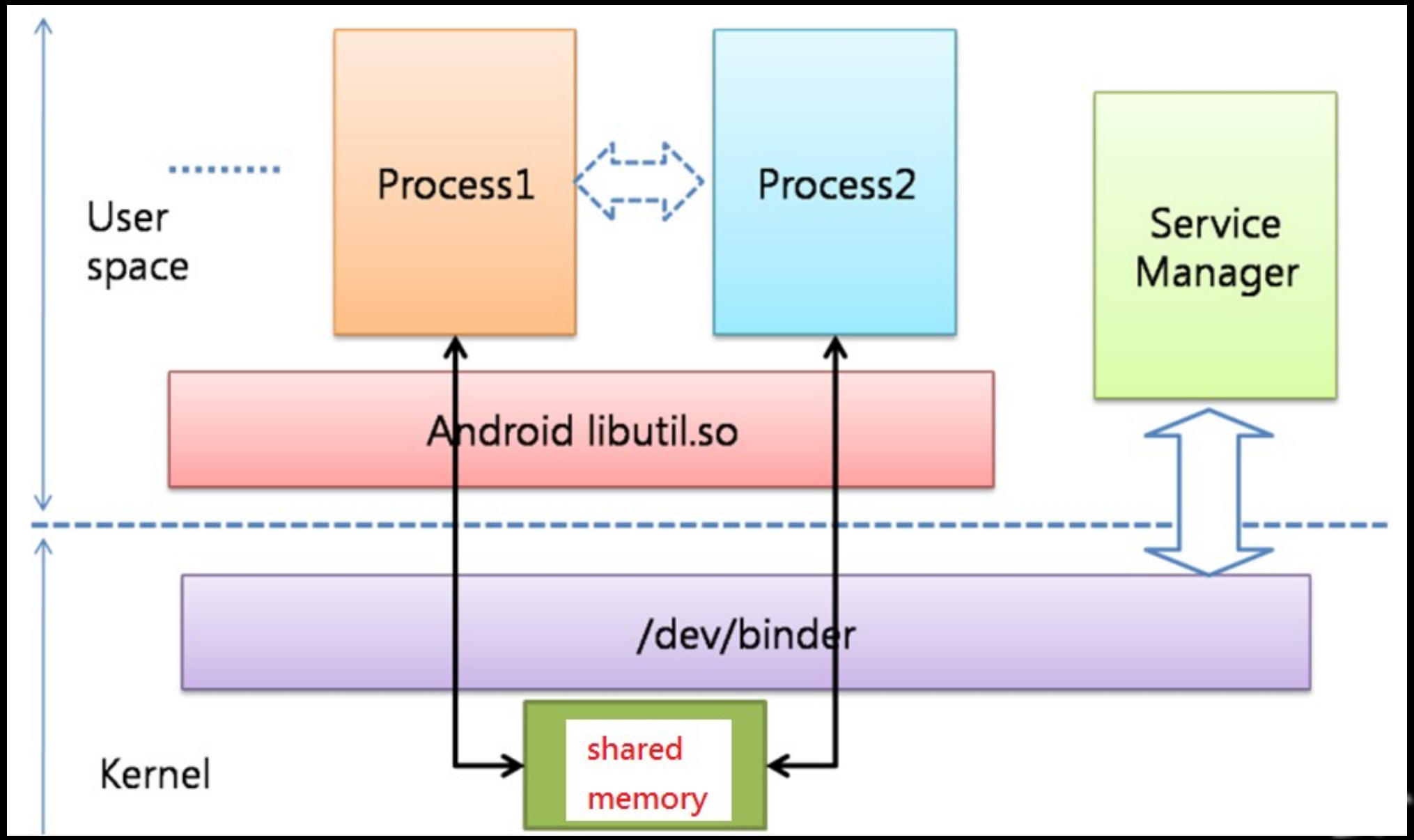


- A pool of threads is associated to each service application to process incoming IPC
- Binder performs mapping of object between two processes.
- Binder uses an object reference as an address in a process's memory space.
- Synchronous call, reference counting

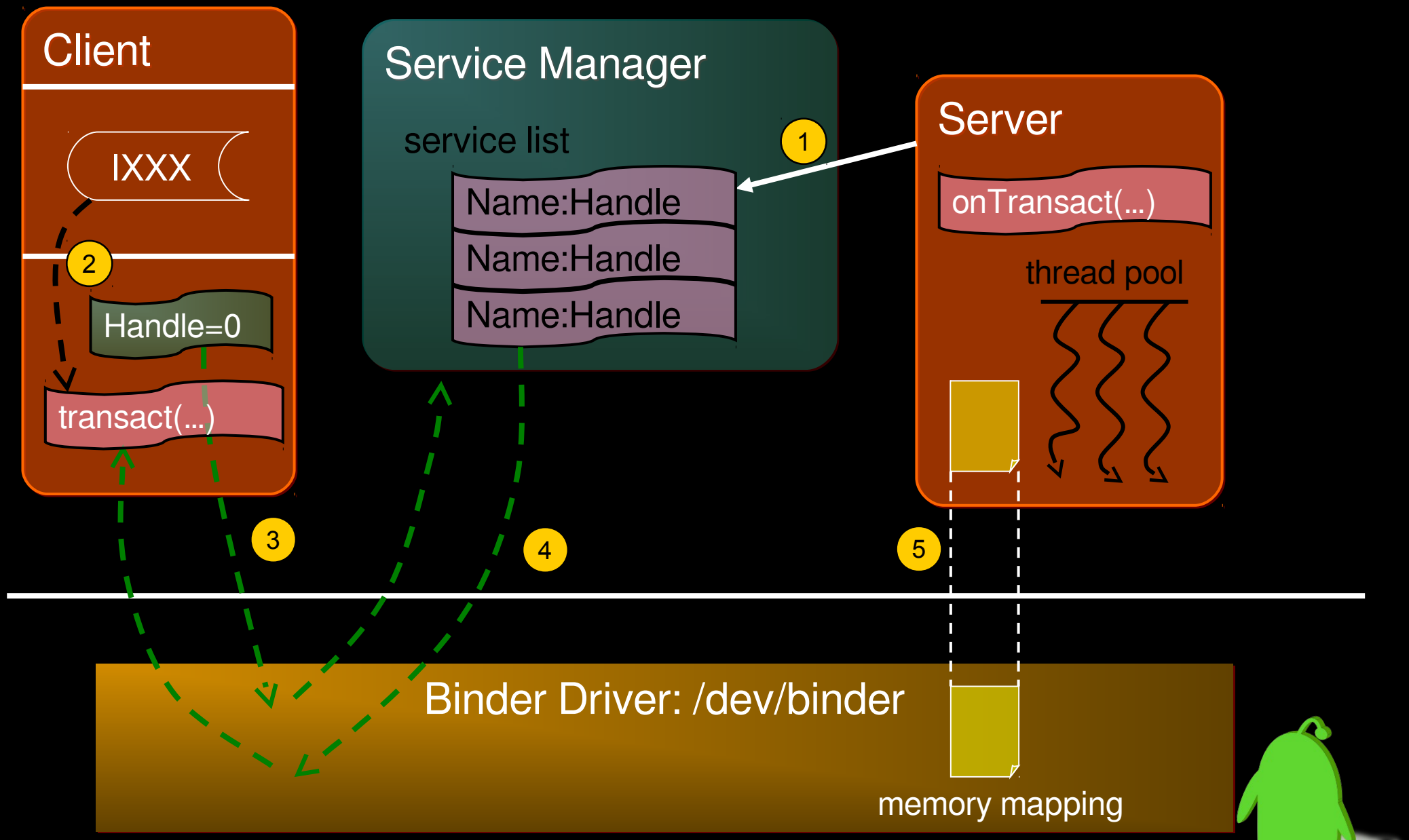
Binder is different from UNIX socket

	socket	binder
internal status	associated to FD	associated to PID (FD can be shared among threads in the same process)
read & write operation	stream I/O	done at once by **ioctl**
network transparency	Yes	No expected local only

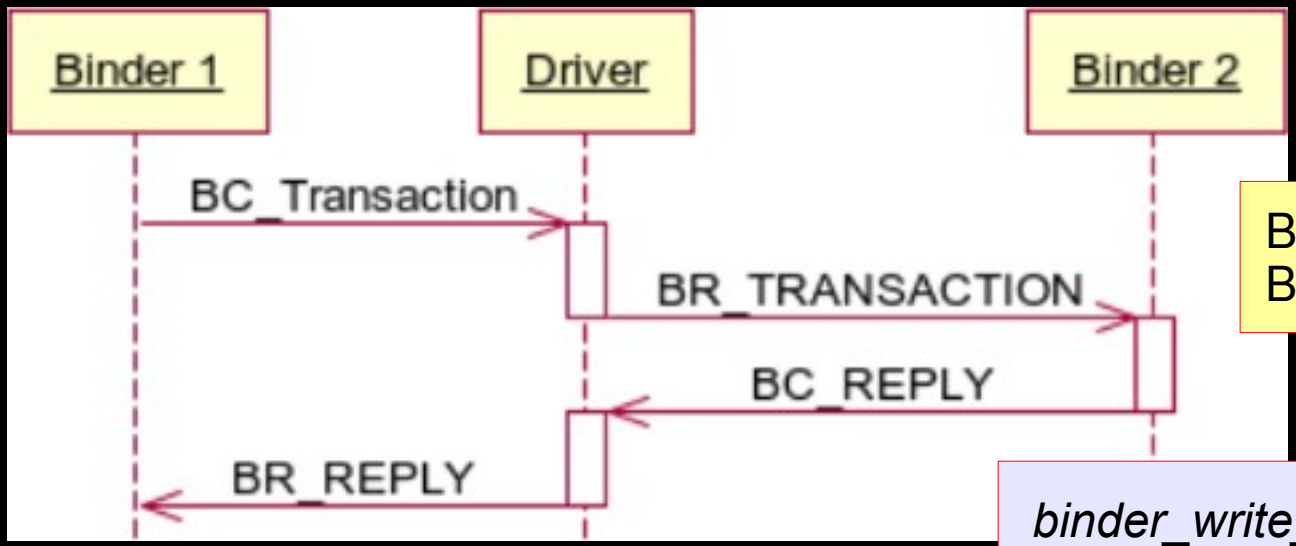
Binder


User space
Process1
Process2
Service Manager
Android libutil.so
/dev/binder
Kernel
shared memory

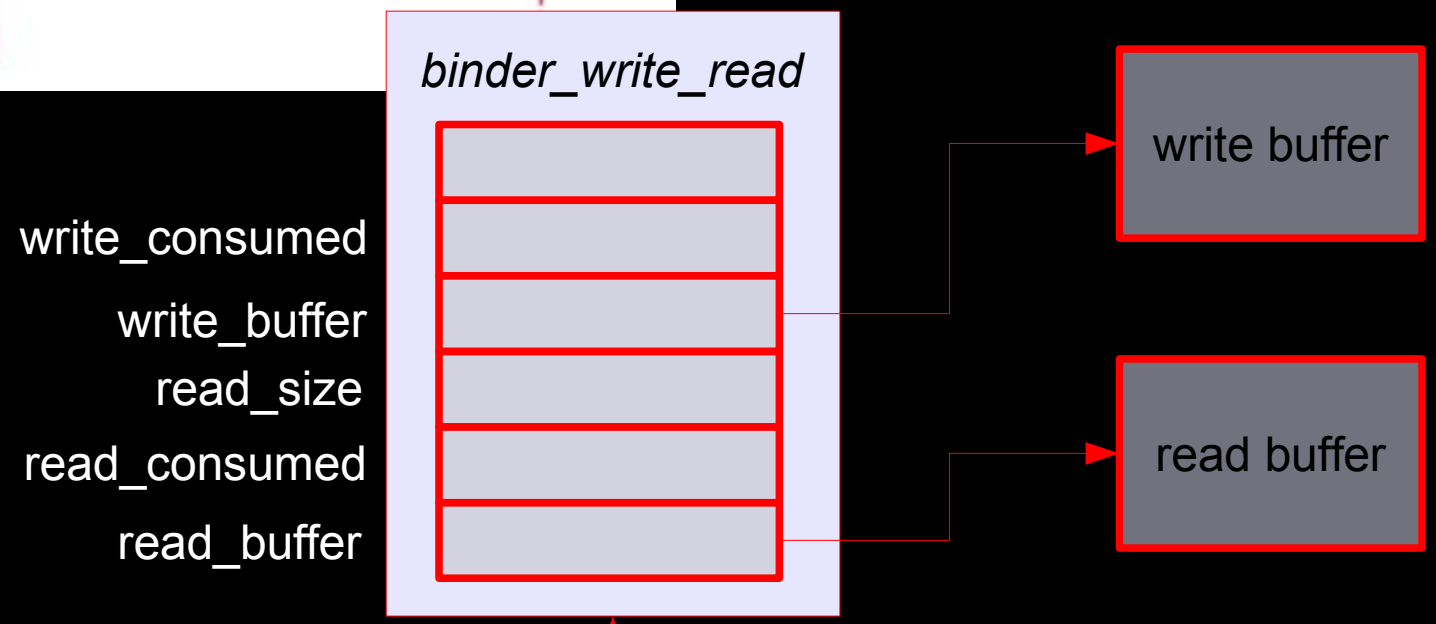
from SM to Binder Driver



Transaction

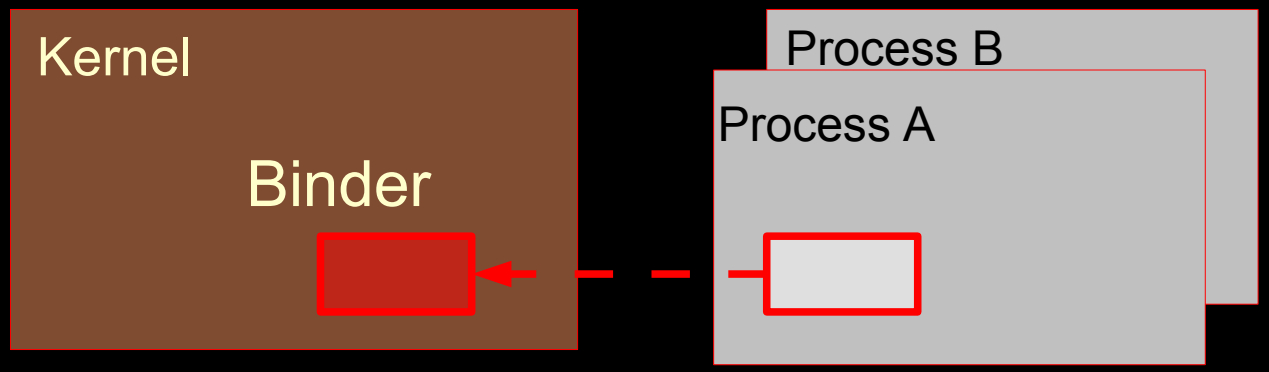


BR → BinderDriverReturnProtocol
BC → BinderDriverCommandProtocol

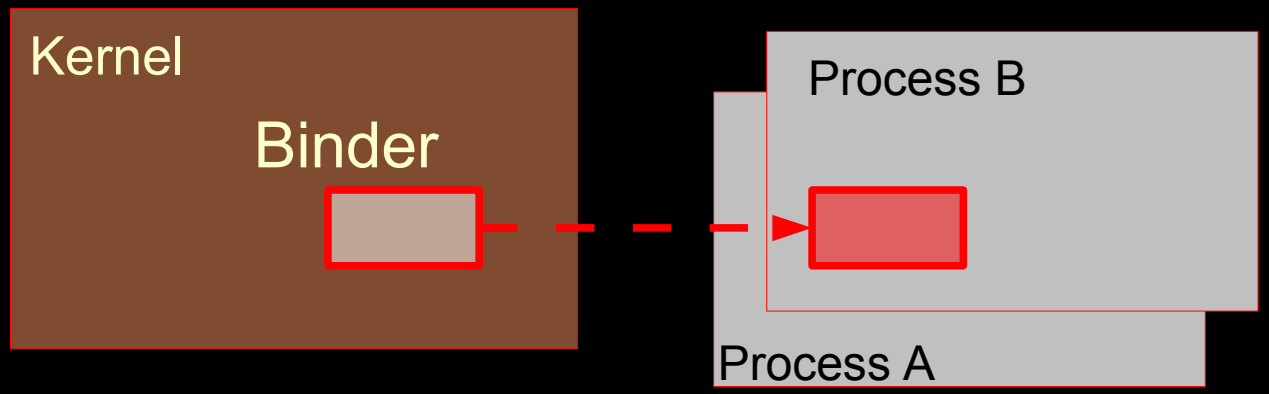



```
if (ioctl(fd, BINDER_WRITE_READ, &bwt ) >= 0)
    err = NO_ERROR;
else
    err = -errno;
```

Transaction of Binder



Copy memory by **copy_from _user**



Copy memory by **copy_to_user**

Internally, Android uses Binder for graphics data transaction across processes.
It is fairly efficient.

Limitation of Binder IPC

- Binders are used to to communicate over process boundaries since different processes don't share a common VM context
  no more direct access to each others Objects (memory).
- Binders are not ideal for transferring large data streams (like audio/video) since every object has to be converted to (and back from) a Parcel.

Binder Performance

- Good
  - Compact method index
  - Native binary marshalling
  - Support of ashmem shortcut
  - No GUID
- Bad
  - Dalvik Parcel overhead
  - ioctl() path is not optimal
  - Interface name overhead
  - Global lock

Binder Security

- Binder's Security Features
  - Securely Determined Client Identity
    - Binder.getCallingUid(), Binder.getCallingPid()
    - Similar to Unix Domain Socket
    - `getsockopt(..., SO_PEERCRED, ...)`
  - Interface Reference Security
    - Client cannot guess Interface Reference
- Service Manager
  - Directory Service for System Services
- Server should check client permission

```
Context.checkPermission(permission, pid, uid)
```

Binder sample program

- Build binder benchmark program

```
cd system/extras/tests/binder/benchmarks
mm
adb push \
  ../../../../out/target/product/crespo/data/nativebenchmark/binderAddInts \
  /data/local/
```

- Execute

```
adb shell
su
/data/local/binderAddInts -d 5 -n 5 &
ps
...
root      17133 16754 4568   860   ffffffff 400e6284 S
/data/local/binderAddInts
root      17135 17133 2520   616   00000000 400e5cb0 R
/data/local/binderAddInts
```

Binder sample program

- Execute

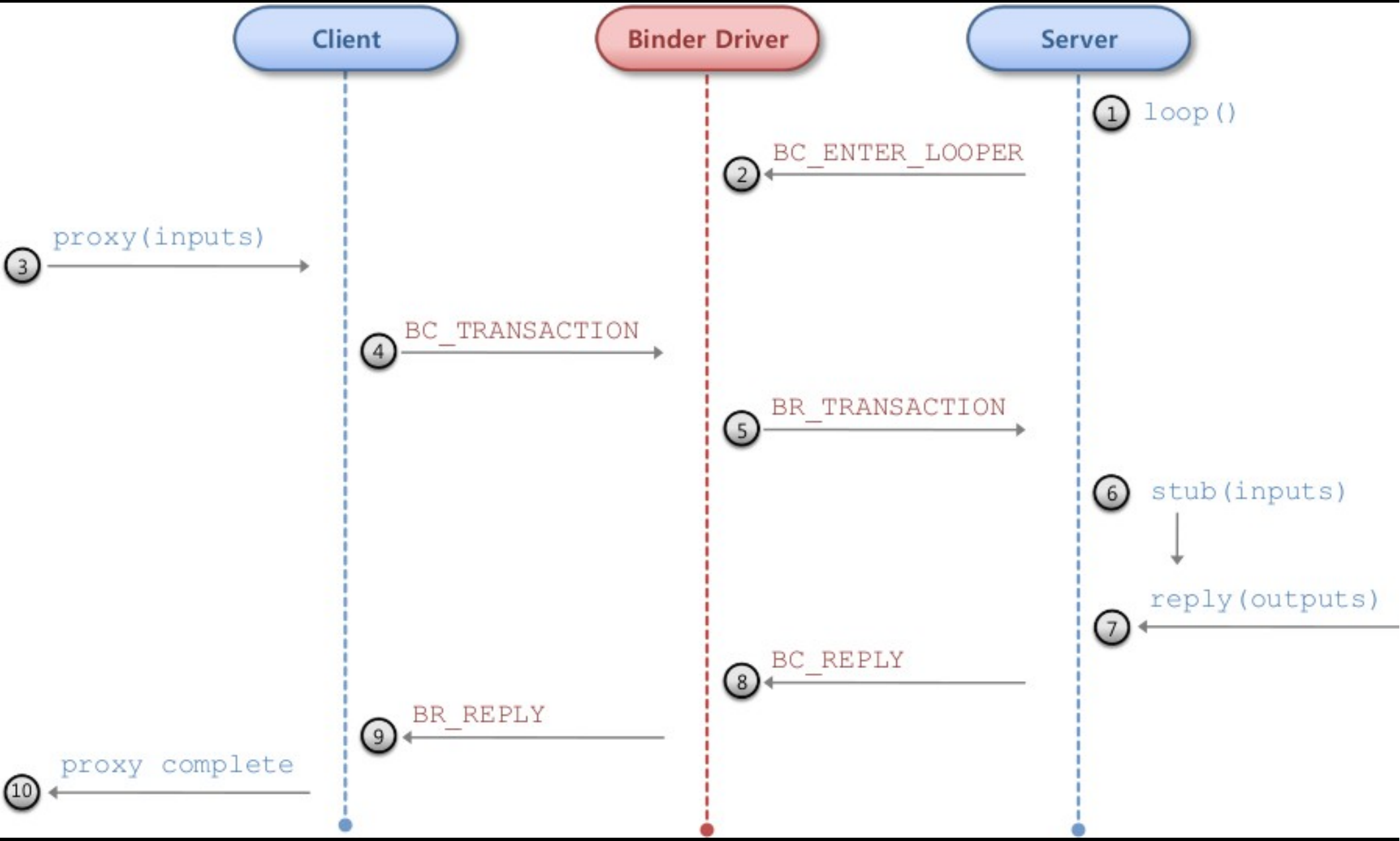
```
/data/local/binderAddInts -d 5 -n 5 &
ps
...
root      17133 16754 4568   860   ffffffff 400e6284 S
/data/local/binderAddInts
root      17135 17133 2520   616   00000000 400e5cb0 R
/data/local/binderAddInts
cat /sys/kernel/debug/binder/transaction_log
transaction_log:3439847: call  from 17133:17133 to 72:0 node
1 handle 0 size 124:4
transaction_log:3439850: reply from 72:72 to 17133:17133 node
0 handle 0 size 4:0
transaction_log:3439855: call  from 17135:17135 to 17133:0
node 3439848 handle 1 size 8:0
...
```

Binder sysfs entries

- **`adb shell ls /sys/kernel/debug/binder`**

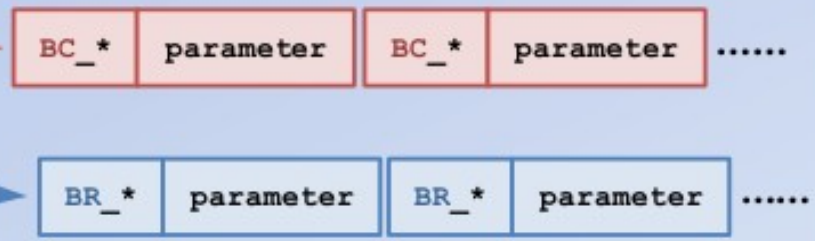
```
failed_transaction_log
proc
state
stats
transaction_log
transactions
```

Remote Procedure Call



BINDER_WRITE_READ

```
struct binder_write_read {
        long            write_size;         /* bytes to write */
        long            write_consumed;     /* bytes consumed by driver */
        unsigned long   write_buffer;
        long            read_size;          /* bytes to read */
        long            read_consumed;      /* bytes consumed by driver */
        unsigned long   read_buffer;
};

#include <sys/ioctl.h>
#include <linux/binder.h>

int binder_write(int fd, void *data, long len) {
        struct binder_write_read bwr;

        bwr.write_size = len;
        bwr.write_consumed = 0;
        bwr.write_buffer = (unsigned) data;
        bwr.read_size = 0;
        bwr.read_consumed = 0;
        bwr.read_buffer = 0;
        return ioctl(fd, BINDER_WRITE_READ, &bwr);
}
```

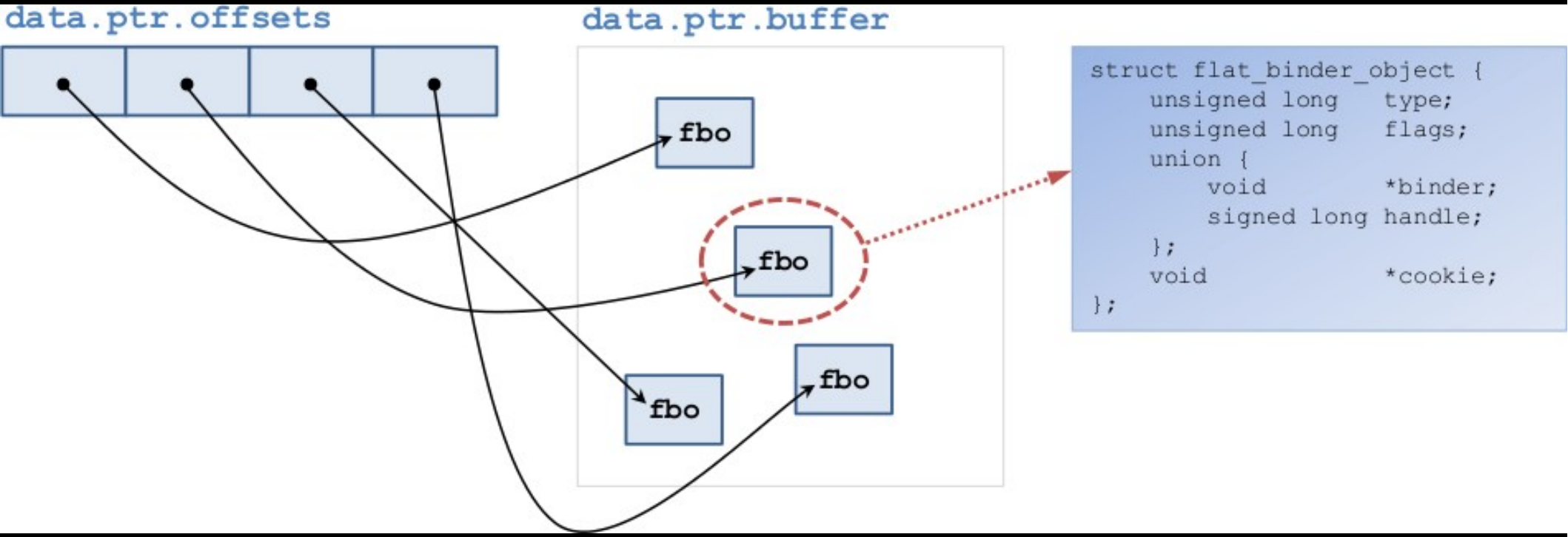


Binder Transaction

- Target Method
  - handle : Remote Interface
  - ptr & cookie : Local Interface
  - code : Method ID
- Parcel - Input/Output Parameters
  - data.ptr.buffer
  - data_size
- Object Reference Management
  - data.ptr.offsets
  - offsets_size
- Security
  - sender_pid
  - sender_euid
- No Transaction GUID
  - Transparent Recursion

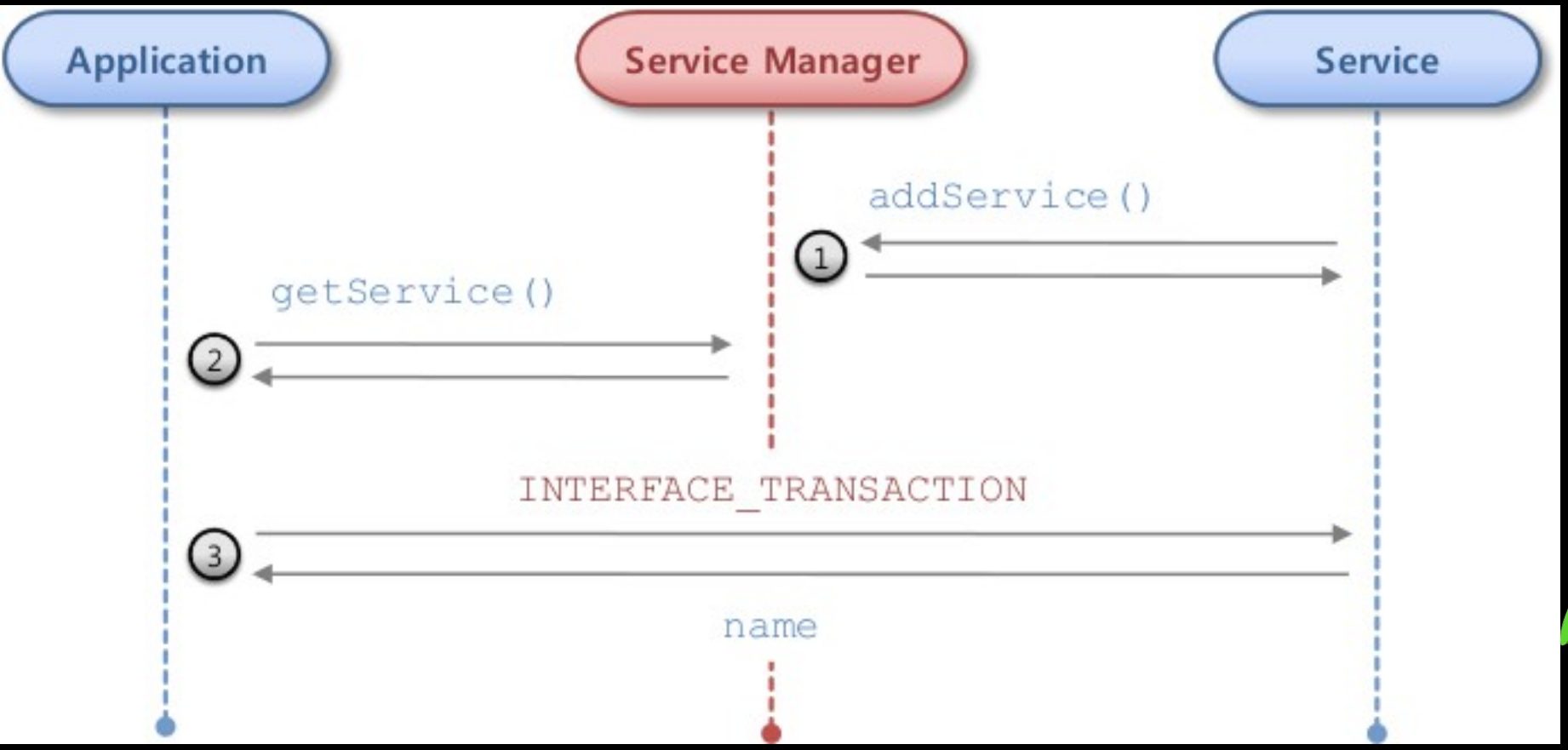
```
#define BC_TRANSACTION
#define BC_REPLY
#define BR_TRANSACTION
#define BR_REPLY

struct binder_transaction_data {
    union {
        size_t          handle;
        void            *ptr;
    } target;
    void                *cookie;
    unsigned int        code;
    unsigned int        flags;
    pid_t               sender_pid;
    uid_t               sender_euid;
    size_t              data_size;
    size_t              offsets_size;
    union {
        struct {
            const void  *buffer;
            const void  *offsets;
        } ptr;
        uint8_t         buf[8];
    } data;
};
```

Object Reference Management



Service Registration and Discovery

- System service is executed by `IServiceManager::addService()` calls.
  Parameter: handle to Binder Driver
- Look up the name of specific service in Binder Driver Map
  `IServiceManager::getService()` returns the handle of the found registered services
- `android.os.IBinder.INTERFACE_TRANSACTION`: the actual name



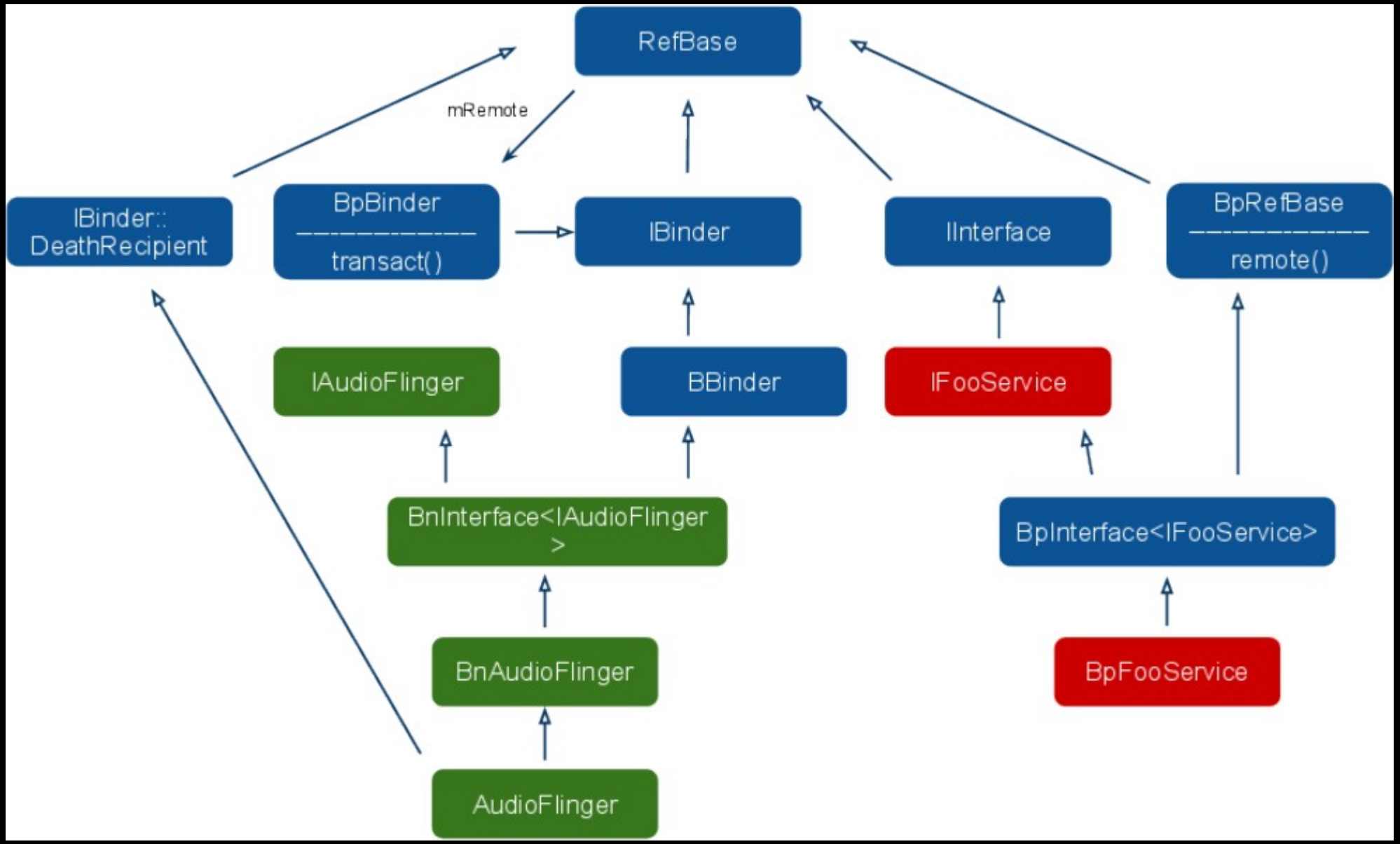
AudioFlinger service
(communicating with Media Server)

- Source: framework/base/media/mediaserver/main_mediaserver.cpp

```
int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    LOGI("ServiceManager: %p", sm.get());
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();
    CameraService::instantiate();
    AudioPolicyService::instantiate();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

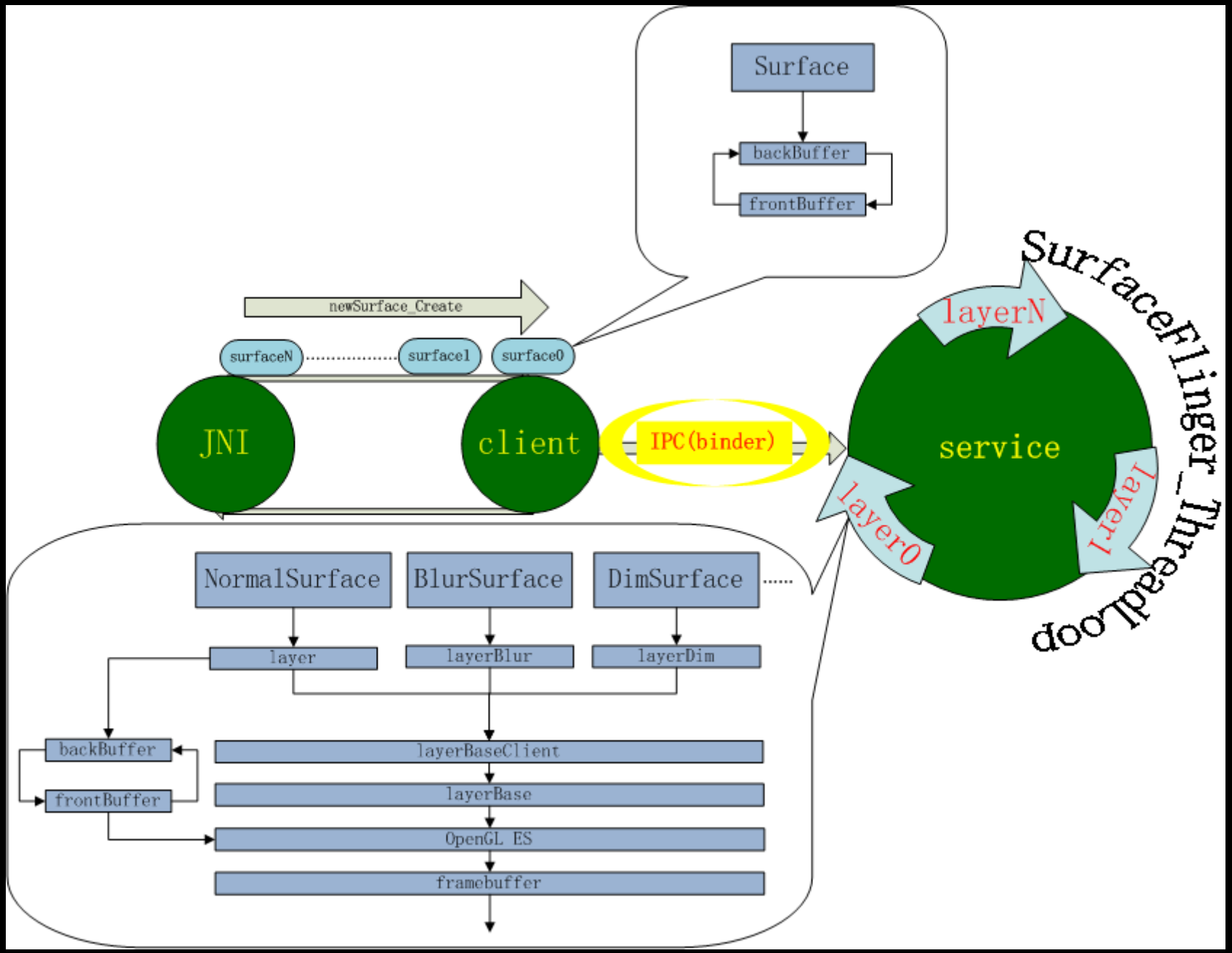
```
void AudioFlinger::instantiate() {
    defaultServiceManager()->addService(
    String16("media.audio_flinger"), new AudioFlinger());
}
```

- AudioFlinger → BnAudioFlinger
- BnAudioFlinger → BnInterface
- BnInterface → BBinder
- BBinder → IBinder



Binder use case: Android Graphics

Real Case



Binder IPC is used for communicating between Graphics client and server.
Taken from http://www.cnblogs.com/xl19862005/archive/2011/11/17/2215363.html

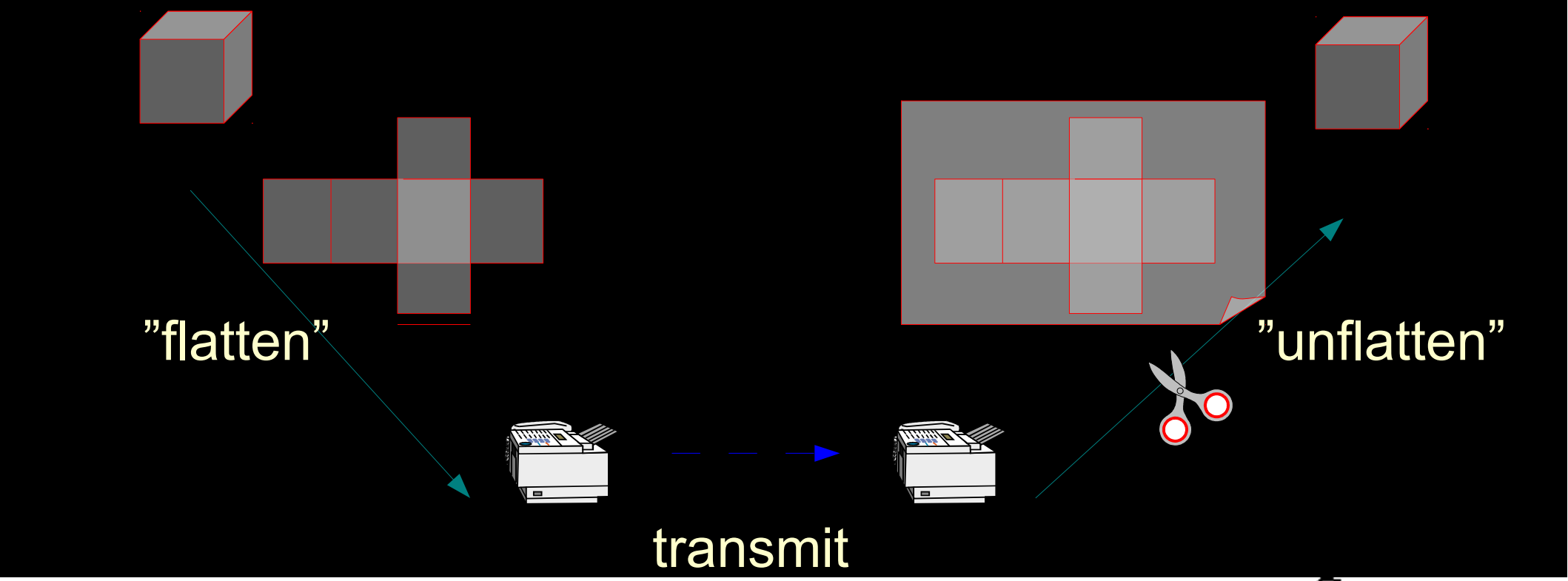
Surface

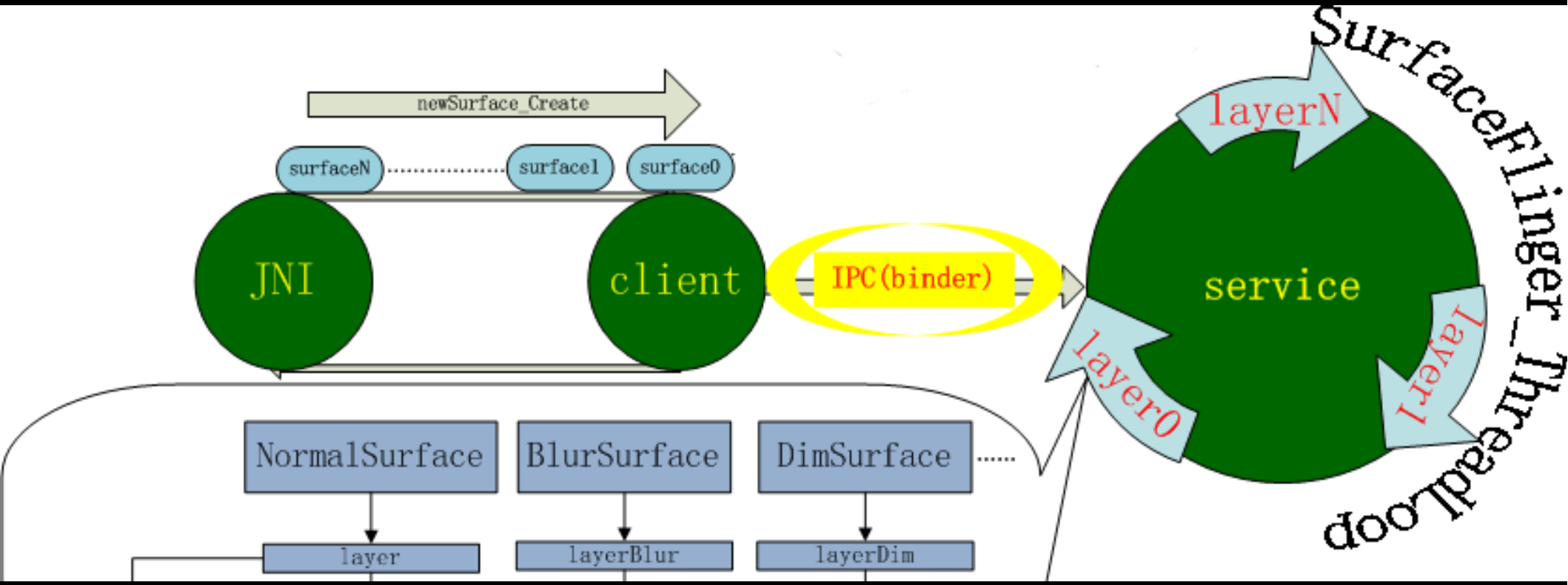
Source: frameworks/base/core/java/android/view/Surface.java

- /* **Handle on to a raw buffer that is being managed by the screen compositor** */

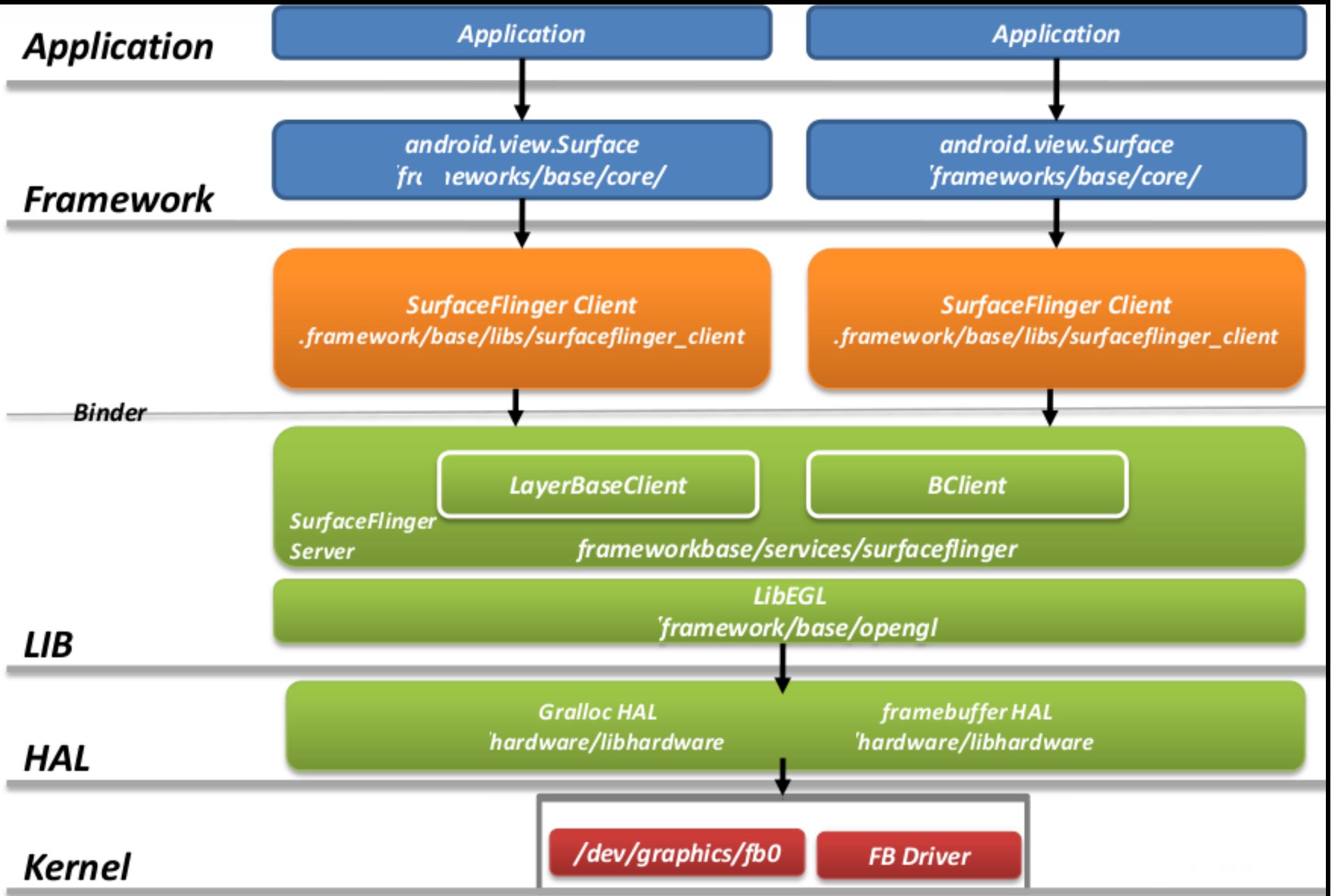
```
public class Surface implements Parcelable {
    public Surface() {
        mCanvas = new CompatibleCanvas();
    }
    private class CompatibleCanvas
                extends Canvas { /* ... */ }
}
```

Surface instances can be written to and restored from a Parcel.

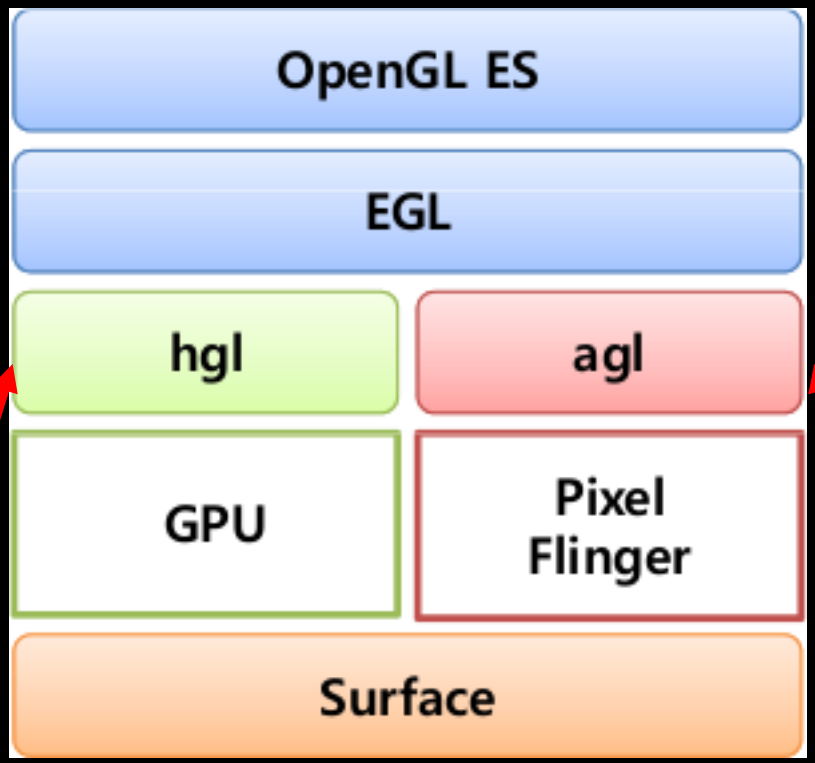

”flatten”
”unflatten”
transmit

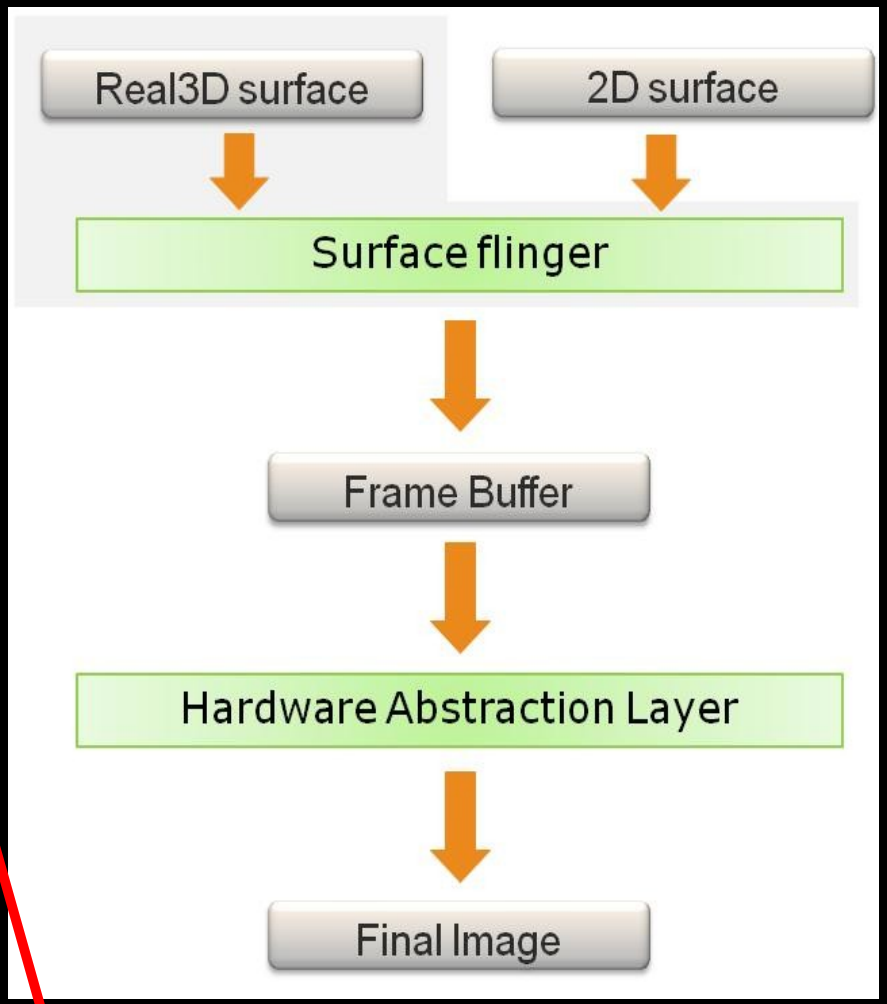

newSurface_Create
surfaceN
surface1
surface0
JNI
client
IPC(binder)
service
layerN
layer1
layer0
SurfaceFlinger_ThreadLoop
NormalSurface
BlurSurface
DimSurface
layer
layerBlur
layerDim

from SurfaceFlinger to Framebuffer



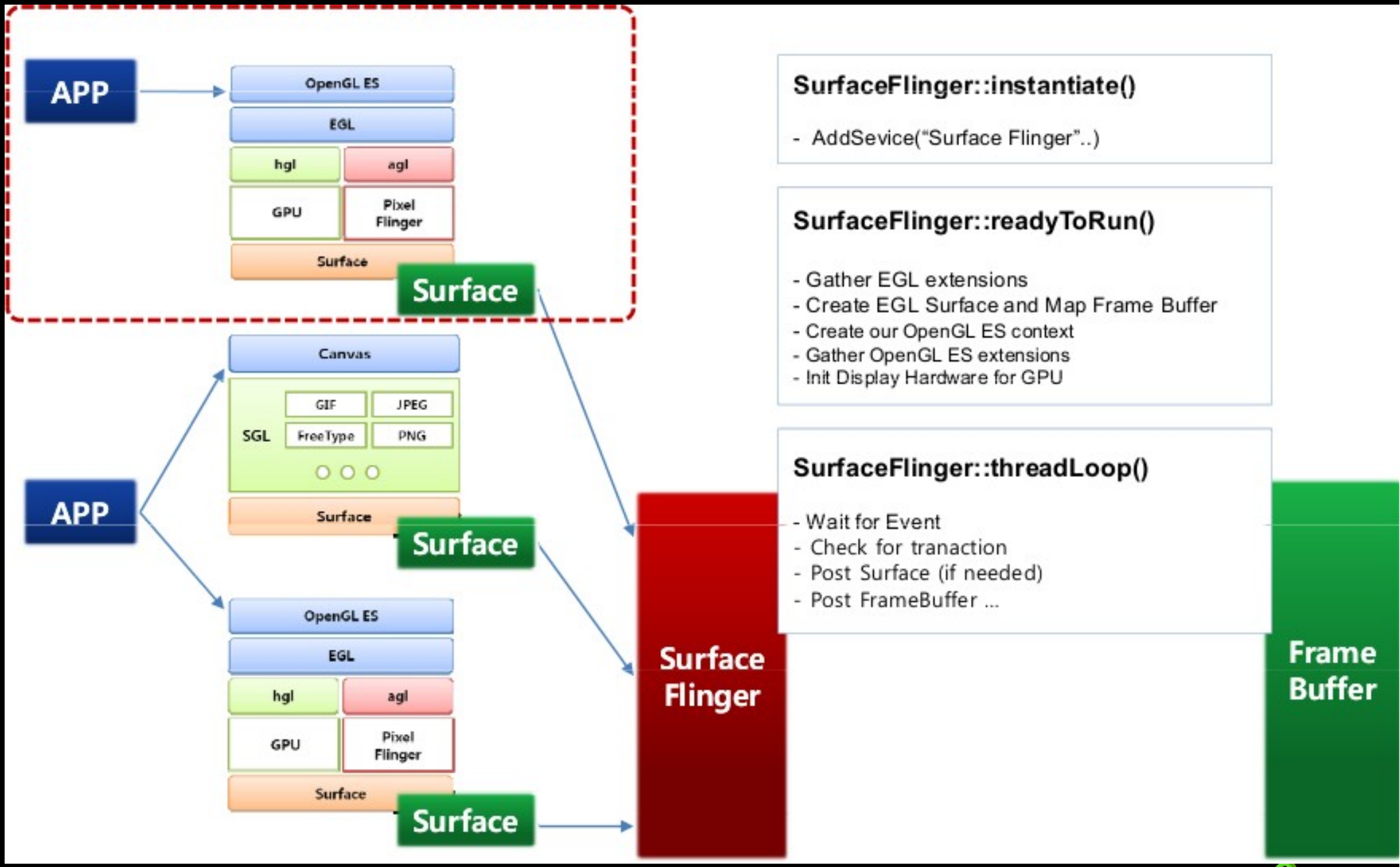
from EGL to SurfaceFlinger





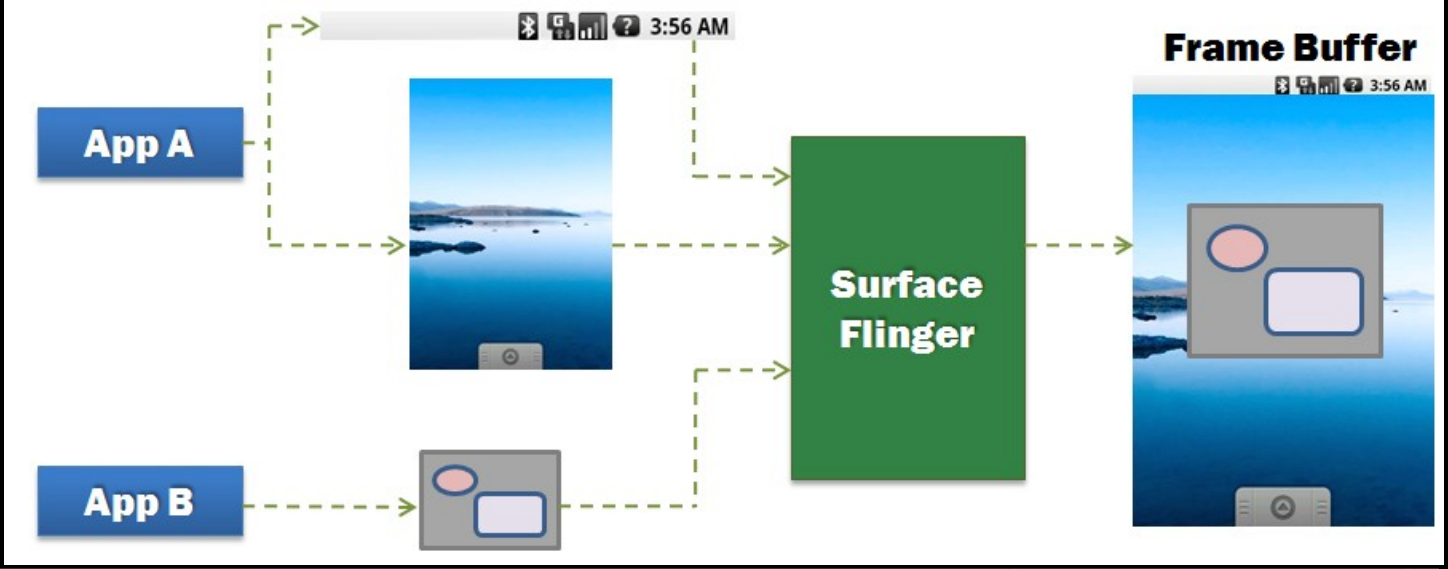
hgl = hardware OpenGL|ES

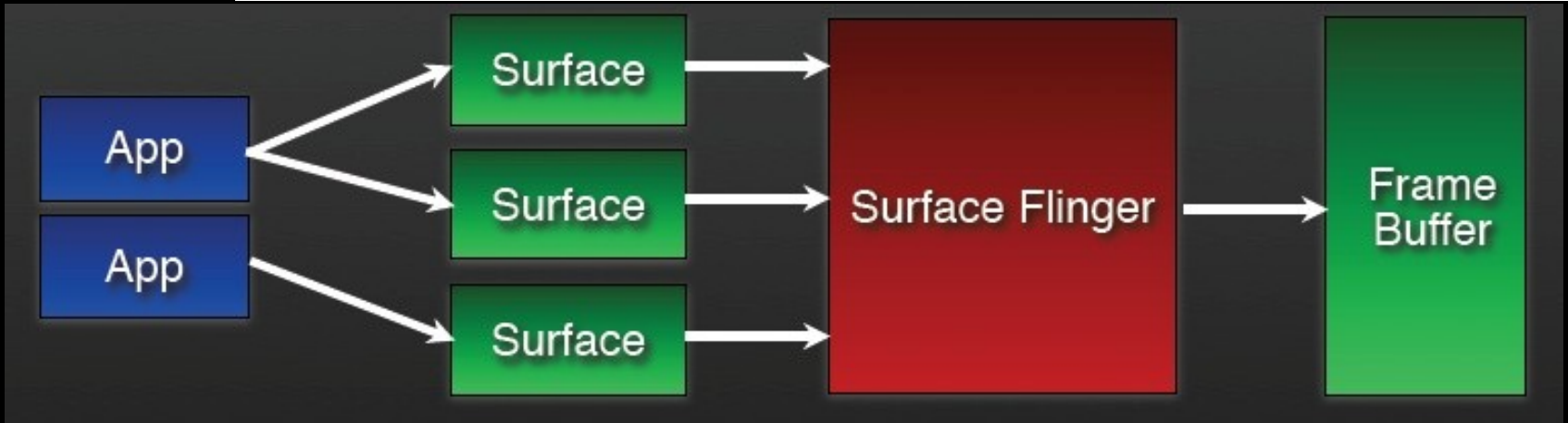
agl = android software OpenGL|ES renderer

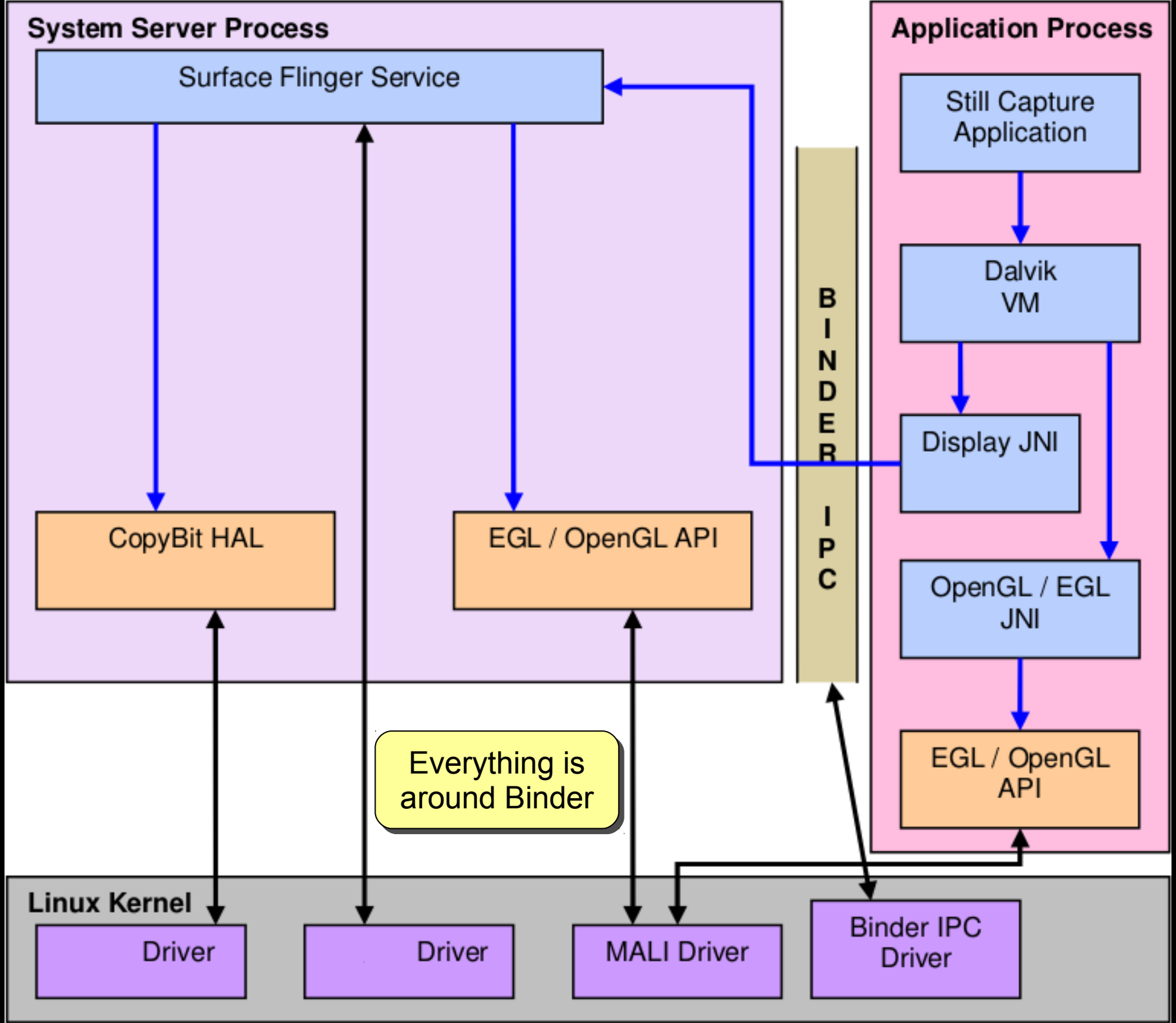

APP
OpenGL ES
EGL
hgl
agl
GPU
Pixel Flinger
Surface
Surface
Canvas
SGL
GIF
JPEG
FreeType
PNG
APP
Surface
Surface
OpenGL ES
EGL
hgl
agl
GPU
Pixel Flinger
Surface
Surface
Surface Flinger
SurfaceFlinger::instantiate()
- AddSevice("Surface Flinger"..)
SurfaceFlinger::readyToRun()
- Gather EGL extensions
- Create EGL Surface and Map Frame Buffer
- Create our OpenGL ES context
- Gather OpenGL ES extensions
- Init Display Hardware for GPU
SurfaceFlinger::threadLoop()
- Wait for Event
- Check for tranaction
- Post Surface (if needed)
- Post FrameBuffer ...
Frame Buffer

Android SurfaceFlinger

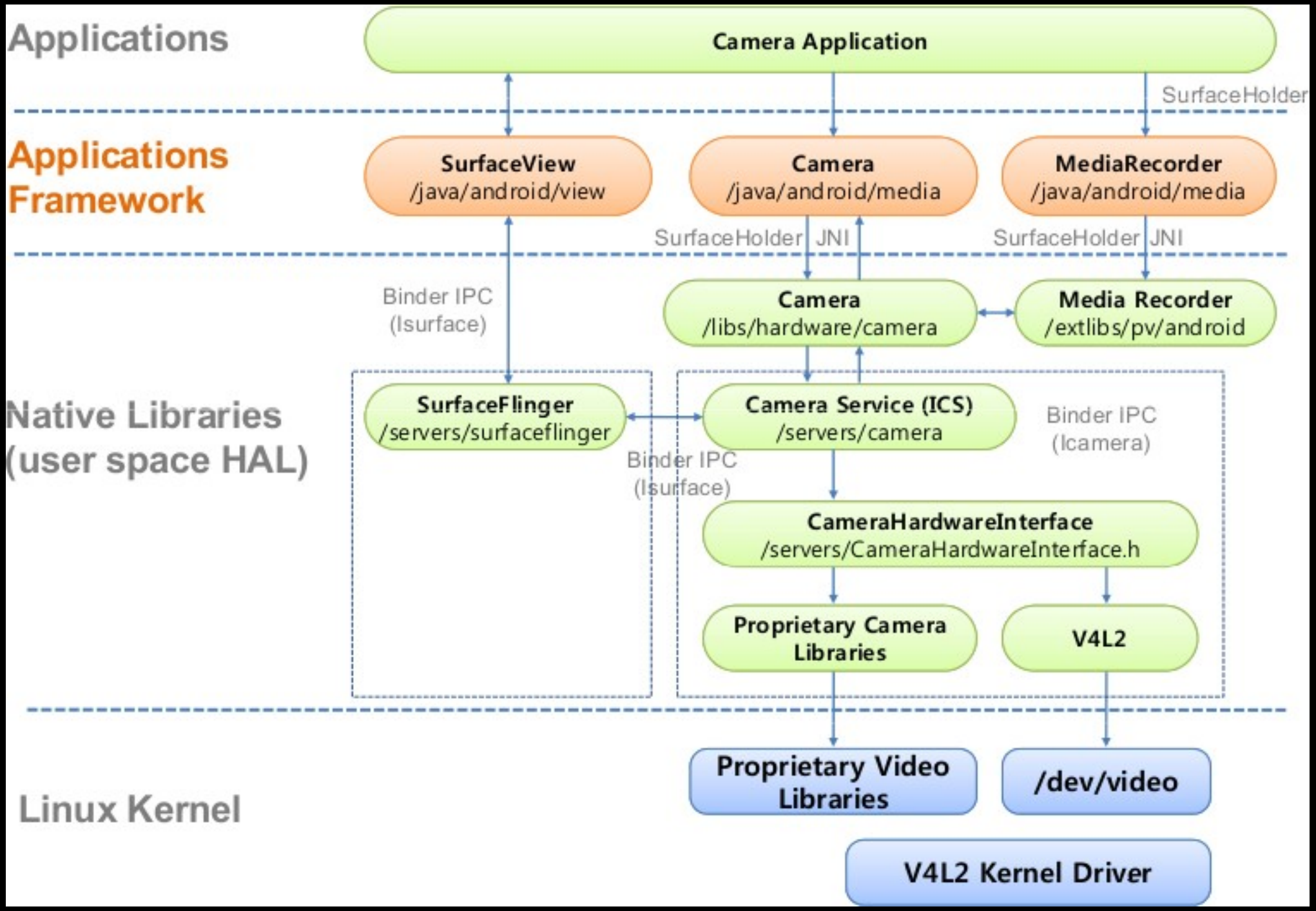
- Properties
  - Can combine 2D/3D surfaces and surfaces from multiple applications
  - Surfaces passed as buffers via Binder IPC calls
  - Can use OpenGL ES and 2D hardware accelerator for its compositions
    - Double-buffering using page-flip






System Server Process
Surface Flinger Service
CopyBit HAL
EGL / OpenGL API
Application Process
Still Capture Application
Dalvik VM
Display JNI
OpenGL / EGL JNI
EGL / OpenGL API
BINDER IPC
Everything is around Binder
Linux Kernel
Driver
Driver
MALI Driver
Binder IPC Driver

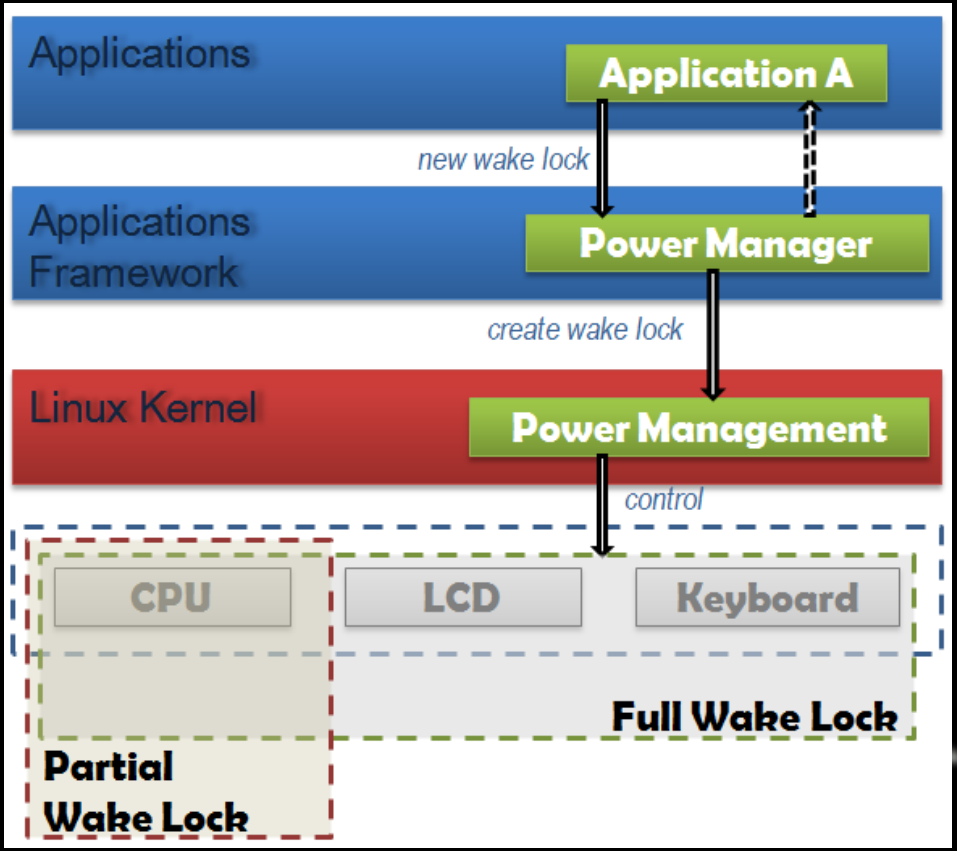
Camera + SurfaceFlinger + Binder



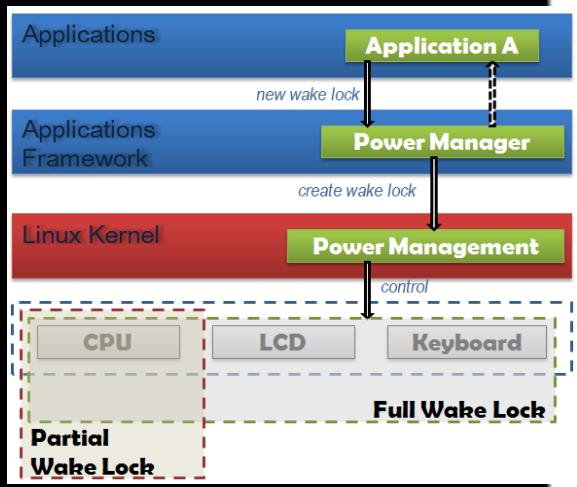
Binder use case: Android Power Management

Android's PM Concepts

- Android PM is built on top of standard Linux Power Management.
- It can support more aggressive PM, but looks fairly simple now.
- Components make requests to keep the power on through "**Wake Locks**".
  - PM does support several types of "Wake Locks".



- If there are no active wake locks, CPU will be turned off.
- If there is are partial wake locks, screen and keyboard will be turned off.



Applications

Application A

Application B

Application C

Wl = newWakeLock(…);
Wl.acquire();
Wl.release();

Applications Framework

PowerManager
Android.os.PowerManager

Power
Android.os.Power

PowerManagerService
Andorid.server.PowerManagerService

JNI

Libraries (user space)

Core Libraries

Power
/lib/hardware/power.c

Linux Kernel

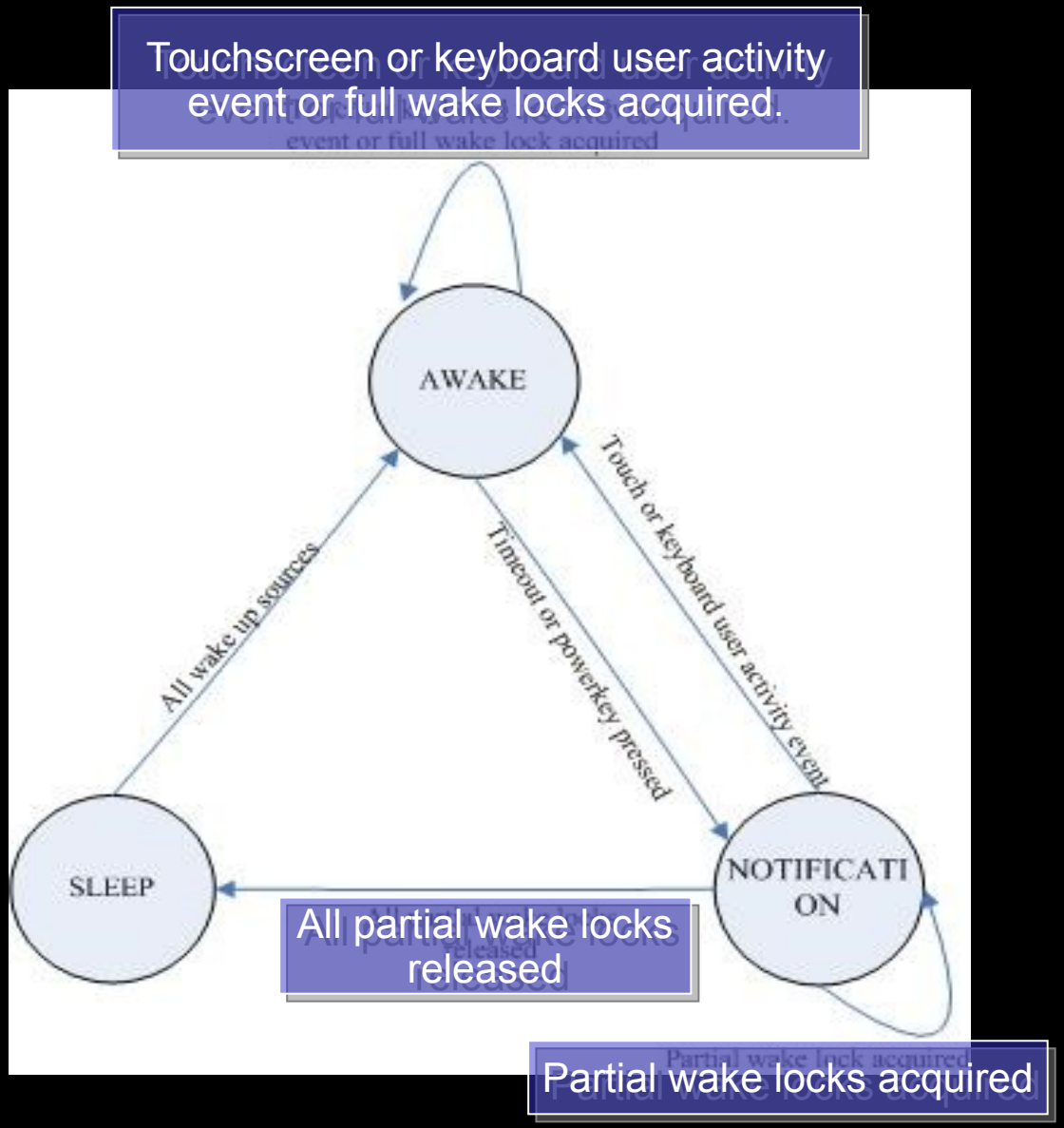
Linux Drivers

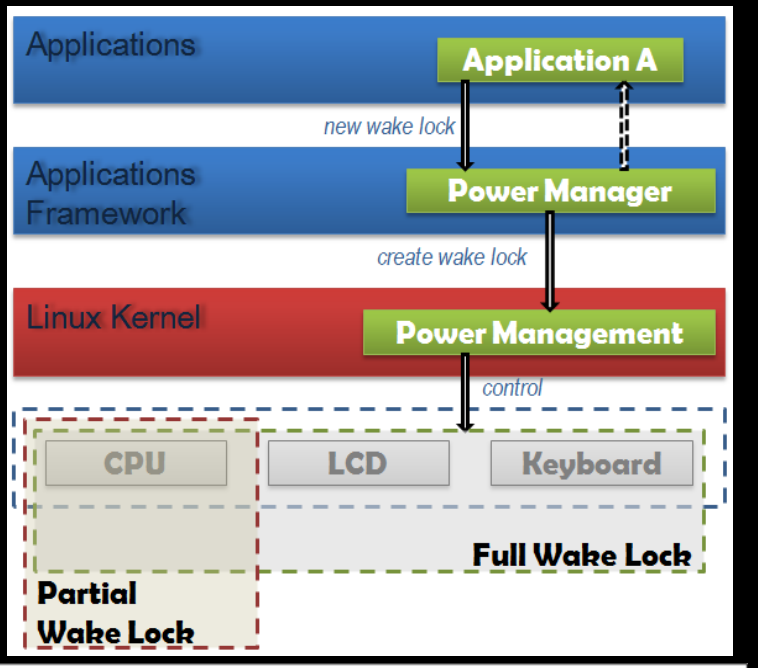
Android Power Management
/drivers/android/power.c

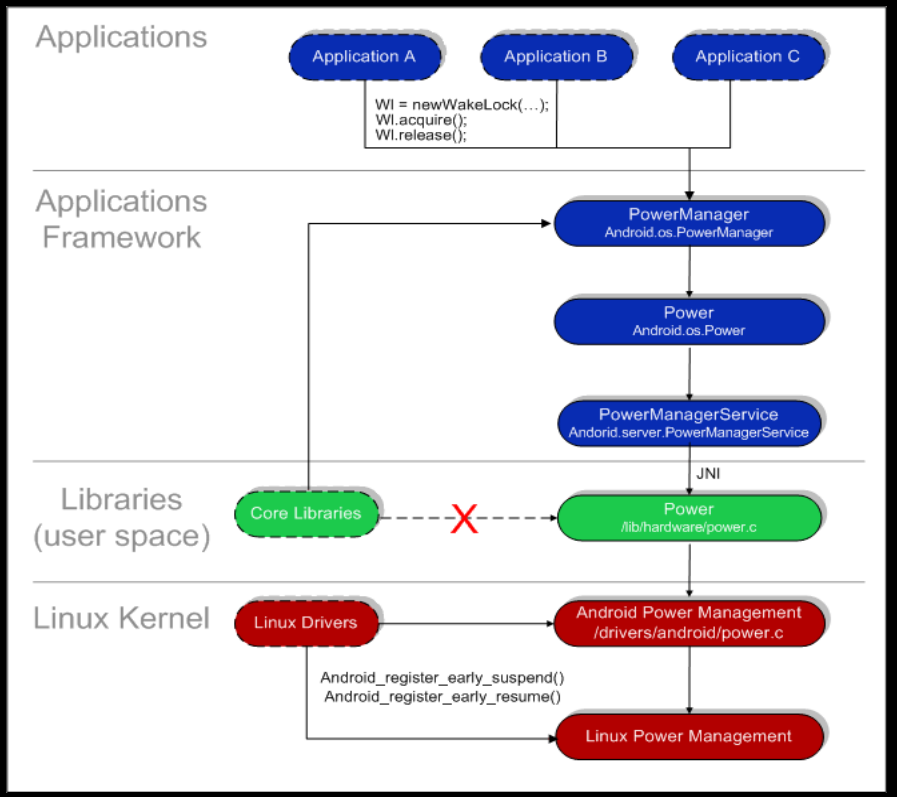
Android_register_early_suspend()
Android_register_early_resume()

Linux Power Management

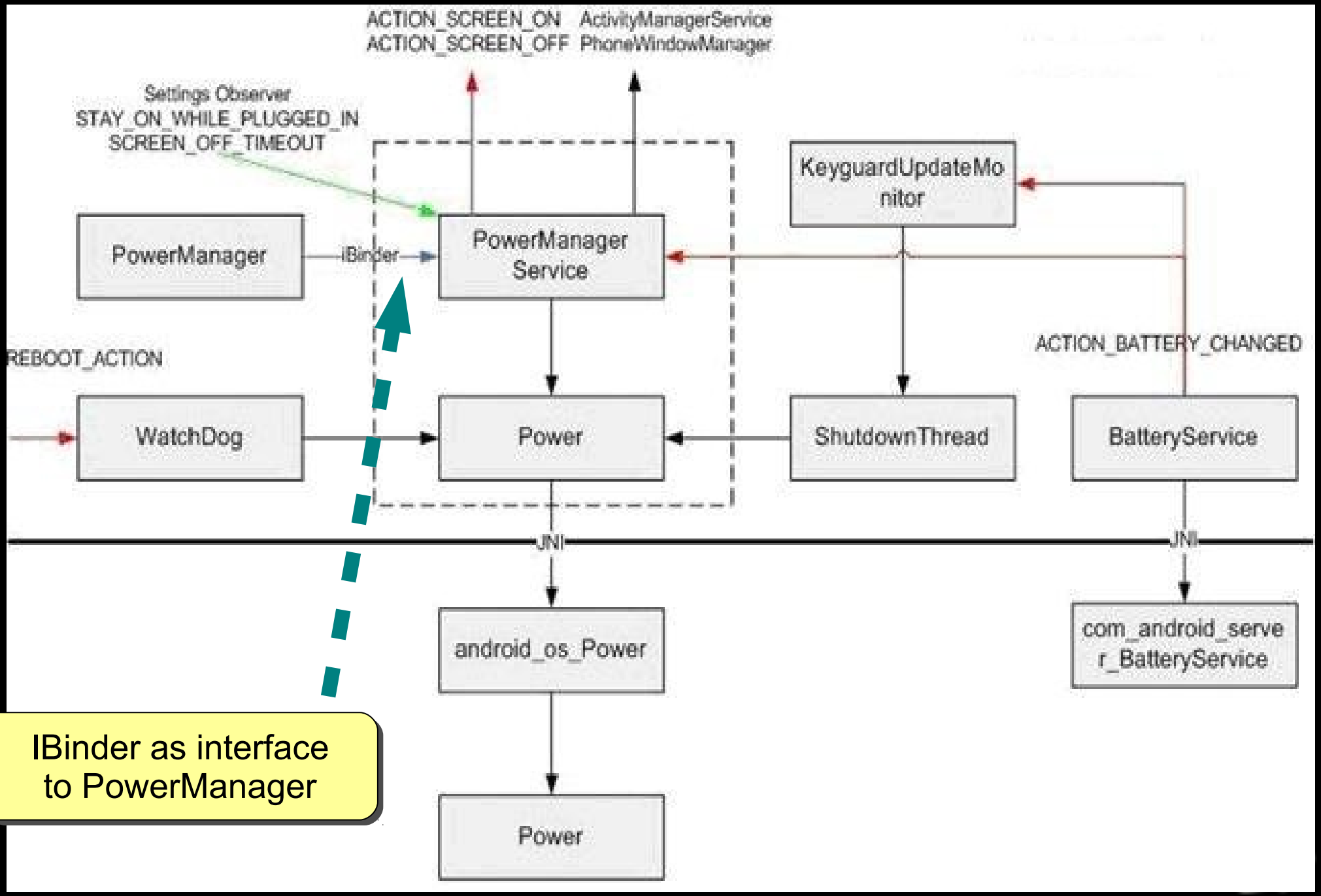
PM State Machine







Design and Implementation



IBinder as interface to PowerManager

Sample WakeLocks usage: AudioFlinger

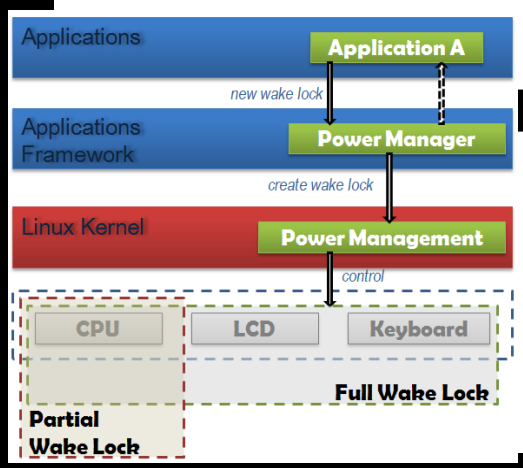
- File frameworks/base/services/audioflinger/AudioFlinger.cpp

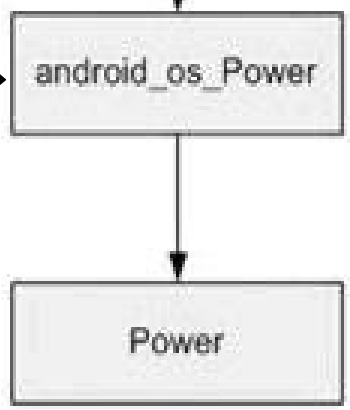
```
void AudioFlinger::ThreadBase::acquireWakeLock_l() {
    if (mPowerManager == 0) {
        sp<IBinder> binder =
            defaultServiceManager()->checkService(String16("power"));
        if (binder == 0) {
            LOGW("Thread %s can't connect to the PM service", mName);
        } else {
            mPowerManager = interface_cast<IPowerManager>(binder);
            binder->linkToDeath(mDeathRecipient);
        }
    }
    if (mPowerManager != 0) {
        sp<IBinder> binder = new BBinder();
        status_t status =
            mPowerManager->acquireWakeLock(POWERMANAGER_PARTIAL_WAKE_LOCK,
                                           binder, String16(mName));
        if (status == NO_ERROR) { mWakeLockToken = binder; }
        LOGV("acquireWakeLock_l() %s status %d", mName, status);
    }
}
```

android_os_Power

frameworks/base/core/jni/android_os_Power.cpp

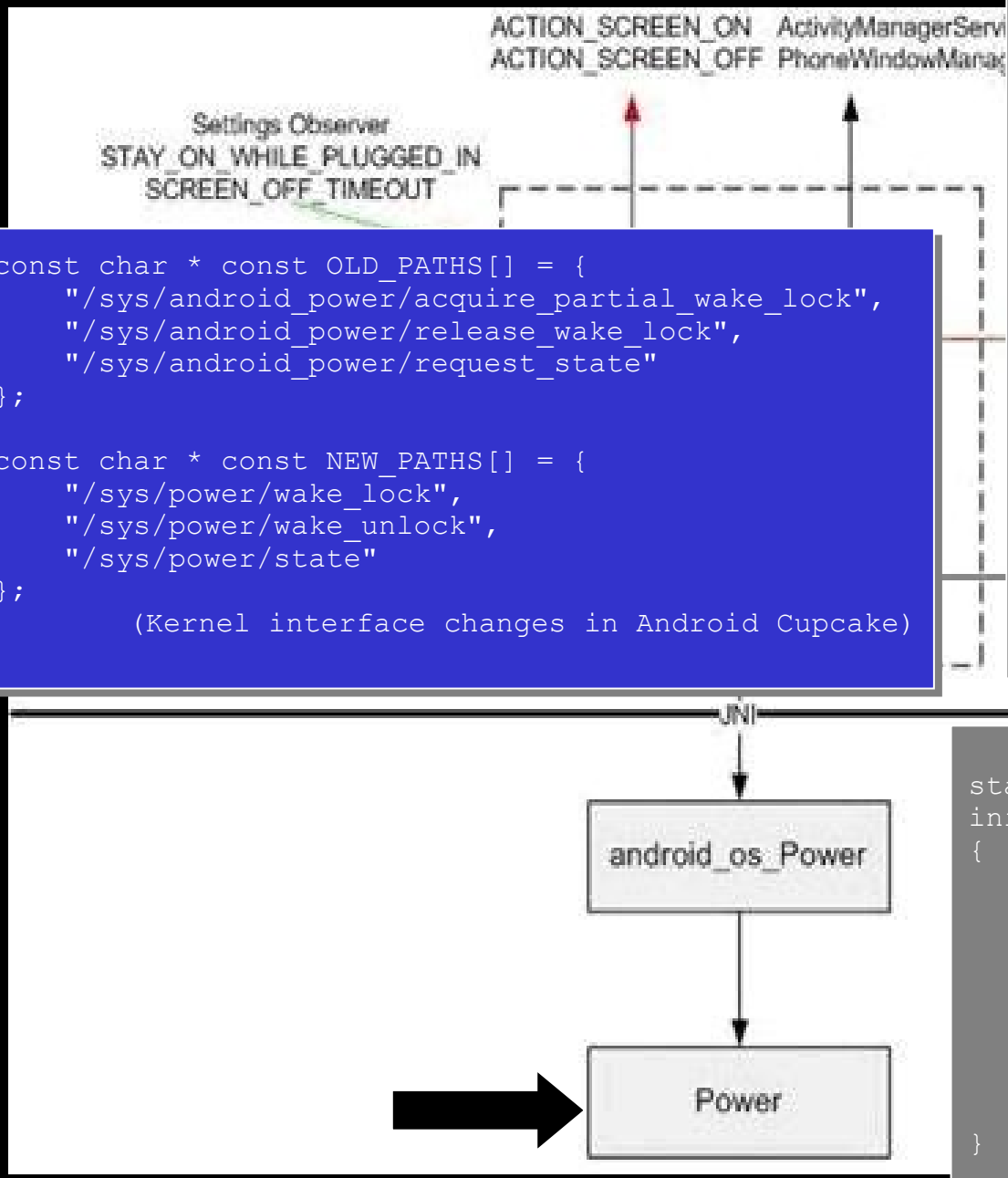
```
...
static JNINativeMethod method_table[] = {
    { "acquireWakeLock", "(ILjava/lang/String;)V", (void*)acquireWakeLock },
    { "releaseWakeLock", "(Ljava/lang/String;)V", (void*)releaseWakeLock },
    { "setLastUserActivityTimeout", "(J)I", (void*)setLastUserActivityTimeout },
    { "setLightBrightness", "(II)I", (void*)setLightBrightness },
    { "setScreenState", "(Z)I", (void*)setScreenState },
    { "shutdown", "()V", (void*)android_os_Power_shutdown },
    { "reboot", "(Ljava/lang/String;)V", (void*)android_os_Power_reboot },
};

int register_android_os_Power(JNIEnv *env)
{
    return AndroidRuntime::registerNativeMethods(
        env, "android/os/Power",
        method_table, NELEM(method_table));
}
```






```
static void
acquireWakeLock(JNIEnv *env, jobject clazz,
                jint lock, jstring idObj)
{
    if (idObj == NULL) {
        throw_NullPointerException(env, "id is null");
        return ;
    }

    const char *id = env->GetStringUTFChars(idObj, NULL);

    acquire_wake_lock(lock, id);

    env->ReleaseStringUTFChars(idObj, id);
}
```

Power




```
const char * const OLD_PATHS[] = {
    "/sys/android_power/acquire_partial_wake_lock",
    "/sys/android_power/release_wake_lock",
    "/sys/android_power/request_state"
};

const char * const NEW_PATHS[] = {
    "/sys/power/wake_lock",
    "/sys/power/wake_unlock",
    "/sys/power/state"
};
        (Kernel interface changes in Android Cupcake)
```

hardware/libhardware_legacy/power/power.c

```
...
int
acquire_wake_lock(int lock, const char* id)
{
    initialize_fds();
    if (g_error) return g_error;

    int fd;
    if (lock == PARTIAL_WAKE_LOCK) {
        fd = g_fds[ACQUIRE_PARTIAL_WAKE_LOCK];
    }
    else {
        return EINVAL;
    }
    return write(fd, id, strlen(id));
}
```

```
static inline void
initialize_fds(void)
{
    if (g_initialized == 0) {
        if(open_file_descriptors(NEW_PATHS) < 0) {
            open_file_descriptors(OLD_PATHS);
            on_state = "wake";
            off_state = "standby";
        }
        g_initialized = 1;
    }
}
```

Android PM Kernel APIs

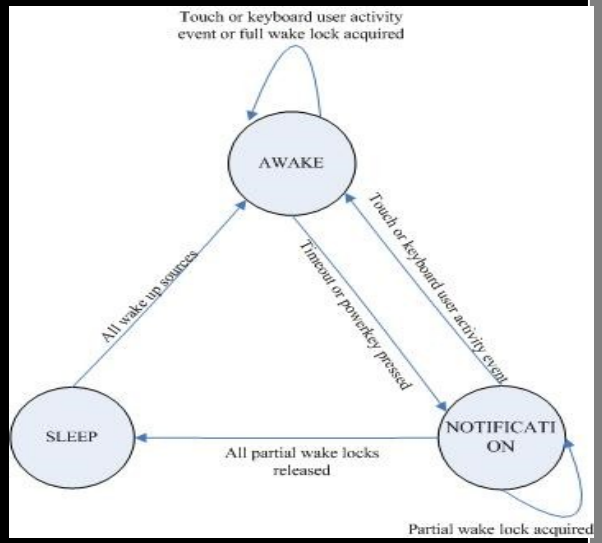
- Source code
  - kernel/power/userwake.c
  - /kernel/power/wakelock.c

```
static int power_suspend_late(
     struct platform_device *pdev,
     pm_message_t state)
{
     int ret =
          has_wake_lock(WAKE_LOCK_SUSPEND) ?
          -EAGAIN : 0;
     return ret;
}

static struct platform_driver power_driver = {
     .driver.name = "power",
     .suspend_late = power_suspend_late,
};
static struct platform_device power_device = {
     .name = "power",
};
```

```
static long has_wake_lock_locked(int type)
{
     struct wake_lock *lock, *n;
     long max_timeout = 0;
     BUG_ON(type >= WAKE_LOCK_TYPE_COUNT);
     list_for_each_entry_safe(lock, n,
               &active_wake_locks[type], link) {
          if (lock->flags & WAKE_LOCK_AUTO_EXPIRE) {
               long timeout = lock->expires - jiffies;
               if (timeout <= 0)
                    expire_wake_lock(lock);
               else if (timeout > max_timeout)
                    max_timeout = timeout;
          } else
               return -1;
     }
     return max_timeout;
}

long has_wake_lock(int type)
{
     long ret;
     unsigned long irqflags;
     spin_lock_irqsave(&list_lock, irqflags);
     ret = has_wake_lock_locked(type);
     spin_unlock_irqrestore(&list_lock, irqflags);
     return ret;
}
```

Android PM Kernel APIs

- kernel/power/wakelock.c




```
static int __init wakelocks_init(void)
{
	int ret;
	int i;

	for (i = 0; i < ARRAY_SIZE(active_wake_locks); i++)
		INIT_LIST_HEAD(&active_wake_locks[i]);


	wake_lock_init(&main_wake_lock, WAKE_LOCK_SUSPEND, "main");
	wake_lock(&main_wake_lock);
	wake_lock_init(&unknown_wakeup, WAKE_LOCK_SUSPEND, "unknown_wakeups");

	ret = platform_device_register(&power_device);
	if (ret) {
		pr_err("wakelocks_init: platform_device_register failed\n");
		goto err_platform_device_register;
	}
	ret = platform_driver_register(&power_driver);
	if (ret) {
		pr_err("wakelocks_init: platform_driver_register failed\n");
		goto err_platform_driver_register;
	}

	suspend_work_queue = create_singlethread_workqueue("suspend");
	if (suspend_work_queue == NULL) {
		ret = -ENOMEM;
		goto err_suspend_work_queue;
	}
```

Frameworks

case study: how the window manager works

Window Manager related

- Applications - need to define onto which display their Activities should go.
- Activity Manager – launch application's activities onto the right display.
- Window Manager – needs to properly handle two Activity stacks and two window stacks
- SurfaceFlinger – needs to handle layer composition for two displays.
- OpenGL / EGL – needs to support one context for each display

Interactions about Window Manager

- Android system service uses a special notification feature of the Binder, that is called link to death mechanism. This facility allows processes to get informed when a Binder of a certain process is terminated.
- In particular, this is the way the Android window manager establishes a link to death relation to the callback Binder interface of each window, to get informed if the window is closed.

startActivity (1)

- Call Stack:
- Processes A, B, C and D. M is the main thread, B is a Binder thread
- A, M: Activity.startActivity (Activity.java)
- A, M: Instrumentation.execStartActivity (Activity.java)
- A, M: ActivityManagerProxy.startActivity (ActivityManagerNative.java)
- B, B: ActivityManagerNative.onTransact(START_ACTIVITY_TRANSACTION) (ActivityManagerNative.java)
- B, B: ActivityManagerService.startActivity (ActivityManagerService.java)
- B, B: ActivityManagerService.startActivityLocked (ActivityManagerService.java) → creates the Activity's HistoryRecord
- B, B: ActivityManagerService.startActivityUnchechedLocked (ActivityManagerService.java)
- B, B: ActivityManagerService.resumeTopActivityLocked (ActivityManagerService.java)
- B, B: ActivityManagerService.startPausingLocked (ActivityManagerService.java)
- B, B: ApplicationThreadProxy.schedulePauseActivity (ApplicationThreadNative.java)

startActivity (2)

- Processes A, B, C and D. M is the main thread, B is a Binder thread
- A, B: ApplicationThreadNative.onTransact(SCHEDULE_PAUSE_ACTIVITY_TRANSACTION) (ApplicationThreadNative.java)
- A, B: ActivityThread.ApplicationThread.schedulePauseActivity (ActivityThread.java)
- A, M: ActivityThread.handleMessage(PAUSE_ACTIVITY) (ActivityThread.java)
- A, M: ActivityThread.handlePauseActivity (ActivityThread.java)
- A, M: ActivityManagerProxy.activityPaused (ActivityManagerNative.java)
- B, B: ActivityManagerNative.onTransact(ACTIVITY_PAUSED_TRANSACTION) (ActivityManagerNative.java)
- B, B: ActivityManagerService.activityPaused (ActivityManagerService.java)
- B, B: ActivityManagerService.completePauseLocked (ActivityManagerService.java)
- B, B: ActivityManagerService.resumeTopActivityLocked (ActivityManagerService.java)
- B, B: ActivityManagerService.startSpecificActivityLocked (ActivityManagerService.java)
- B, B: ActivityManagerService.startProcessLocked (ActivityManagerService.java) → creates the ProcessRecord
- B, B: Process.start (ActivityManagerService.java)
- B, B: Process.startViaZygote (Process.java)
- B, B: Process.zygoteSendArgsAndGetPid (Process.java)
- B, B: Process.openZygoteSocketIfNeeded (Process.java) → send a request using Zygote's local socket to fork a new child process
- ... at this point the call stacks of Zygote and Binder are executed.

startActivity (3)

- Processes A, B, C and D. M is the main thread, B is a Binder thread
- State: Continue at ActivityThread.main which is executed in the forked Zygote child process.
- D, M: ActivityThread.main (ActivityThread.java)

  → creates the ActivityThread instance for this process

  → each ActivityThread contains the ApplicationThread instance for the process which manages activity and service lifecycles etc.

  → the ApplicationThread is responsible for managing the software component container (process) on behalf of the ActivityManagerService
- D, M: ActivityThread.attach (ActivityThread.java)
- D, M: RuntimeInit.setApplicationObject (RuntimeInit.java)
- D, M: ActivityManagerNative.getDefault

  (ActivityManagerNative.java) → connects back to ActivityManagerService
- D, M: ActivityManagerProxy.attachApplication

  (ActivityManagerNative.java)

  → registers the ActivityThread's ApplicationThread service object at the ActivityManagerService

startActivity (4)

- Processes A, B, C and D. M is the main thread, B is a Binder thread
- State: Continue at ActivityThread.main which is executed in the forked Zygote child process.
- B, B: ActivityManagerNative.onTransact(ATTACH_APPLICATION_TRANSACTION)
  (ActivityManagerNative.java) → gets an endpoint reference to the ActivityThreads' ApplicationThread service object (ApplicationThreadProxy)
- B, B: ActivityManagerService.attachApplication
  (ActivityManagerService.java)
- B, B: ActivityManagerService.attachApplicationLocked
  (ActivityManagerService.java)
- B, B: ApplicationThreadProxy.bindApplication
  (ApplicationThreadNative.java)
- B, B: ActivityManagerService.realStartActivityLocked
  (ActivityManagerService.java) → uses ProcessRecord and HistoryRecord
- B, B: ApplicationThreadProxy.scheduleLaunchActivity
  (ApplicationThreadNative.java) → ApplicationThreadProxy.bindApplication and ApplicationThreadProxy.scheduleLaunchActivity are called next to each other. But B1 and B2 do NOT execute in parallel inside of process C. So B2 does not start before B1 is done with ActivityThread.ApplicationThread.bindApplication.

startActivity (5)

- Processes A, B, C and D. M is the main thread, B is a Binder thread
- D, B1: ApplicationThreadNative.onTransact (BIND_APPLICATION_TRANSACTION)
  (ApplicationThreadNative.java)
- D, B1: ActivityThread.ApplicationThread.bindApplication
  (ActivityThread.java)
- D, M: ActivityThread.handleMessage(BIND_APPLICATION)
  (ActivityThread.java)
- D, M: ActivityThread.handleBindApplication
  (ActivityThread.java)
- D, M: ActivityThread.PackageInfo.makeApplication
  (ActivityThread.java)

  → creates the initial app's ApplicationContext

  → this ApplicationContext is returned by Activity.getApplicationContext
- D, M: Instrumentation.newApplication
  (android.app.Application)

startActivity (6)

- Processes A, B, C and D. M is the main thread, B is a Binder thread
- D, B2: ApplicationThreadNative.onTransact (SCHEDULE_LAUNCH_ACTIVITY_TRANSACTION) (ApplicationThreadNative.java)
- D, B2: ActivityThread.ApplicationThread.scheduleLaunchActivity (ActivityThread.java)
- D, M: ActivityThread.handleMessage(LAUNCH_ACTIVITY) (ActivityThread.java)
- D, M: ActivityThread.handleLaunchActivity (ActivityThread.java)
- D, M: ActivityThread.performLaunchActivity (ActivityThread.java)

  → creates the Activity object
- D, M: Instrumentation.newActivity (Instrumentation.java)
- D, M: ActivityThread.PackageInfo.makeApplication (ActivityThread.java) → returns existing app object
- D, M: create new ApplicationContext for the newly created Activity
- D, M: Activity.attach (Activity.java) → gets the newly created ApplicationContext as argument

  → creates the Activity's window and attaches it to its local WindowManager
- D, M: Instrumentation.callActivityOnCreate (Instrumentation.java)
- D, M: Activity.onCreate (Activity.java) → Activity.setContentView
- D, M: Activity.performStart (Activity.java)
- D, M: Activity.onStart (Activity.java)

startActivity (7)

- Processes A, B, C and D. M is the main thread, B is a Binder thread
- D, M: Instrumentation.callActivityOnRestoreInstanceState (Instrumentation.java)
- D, M: Activity.onRestoreInstanceState (Activity.java)
- D, M: Instrumentation.callActivityOnPostCreate (Instrumentation.java)
- D, M: Activity.onPostCreate (Activity.java)
- D, M: ActvityThread.handleResumeActivity (ActivityThread.java)
- D, M: ActivityThread.performResumeActivity (ActivityThread.java)
- D, M: Activity.performResume (Activity.java)
- D, M: Instrumentation.callActivityOnResume (Instrumentation.java)
- D, M: Activity.onResume (Activity.java)
- D, M: Activity.onPostResume (Activity.java)
- D, M: Window.LocalWindowManager.addView (Window.java)

  → attaches the Activity's view hierarchy to the WindowManagerService via the local WindowManager
- D, M: Activity.makeVisible (Activity.java)

Review

- Low-level parts
- Process, Thread, system call
- Memory operations
- Binder IPC
- interactions with frameworks

Reference

- Deep Dive into Android IPC/Binder Framework, Aleksandar (Saša) Gargenta
- Inter-process communication of Android, Tetsuyuki Kobayashi
- 淺談 Android 系統進程間通信 (IPC) 機制 Binder 中的 Server 和 Client 獲得 Service Manager 接口之路

  http://blog.goggb.com/?post=1580
- Service 與 Android 系統設計，宋寶華
- Android Binder – Android Interprocess Communication, Thorsten Schreiber
- Kdbus Details, Greg Kroah-Hartman
- MemChannel: An Efficient Inter-Process Communication
- Mechanism for Mobile Sensing Applications

http://0xlab.org