ANDROID ARCHITECTURE AND BINDER

DHINAKARAN PANDIYAN, SAKETH PARANJAPE

1. INTRODUCTION

"Android is a Linux-based operating system designed primarily for touchscreen mobile devices such as smartphones and tablet computers" [1]. Android is also the world's most widely used smartphone platform having the highest market share. The platform is made of the open source operating system Android, devices that run it and an extensive repository of applications developed for the OS. The open source nature of the OS allows anybody to freely modify and distribute it. The OS has code written in Assembly, Java, C++ and C. The applications are primarily written in Java; however C++ can also be used. Google PlayStore, the application repository for Android, has more than 700,000 applications. The devices that run Android can have the hardware range from a bare minimum 32MB RAM, 200 MHz processor to those with a state-of-the-art Quad core processor and 2GB of RAM. The primary hardware platform is the ARM architecture.

Android's kernel was based on the Linux 2.6 kernel. The latest version of Android called Jelly Bean has the Linux kernel 3.0.3. Android has added new drivers to the kernel, middleware, libraries and an extensive set of APIs that aid application development on top of the kernel. The relevance of smartphones in today's world, ubiquity of the Android operating system and intriguing customization of Linux to run on mobile devices make a compelling case to study Android. Through our project we tried to explore the Android software stack and its unique features, primarily the Inter Process Communication (IPC) mechanism – Binder. Binder is a key and complex component that is essential to the functioning of the Android operating system. This report is a documentation of our learning. Understanding the data flow, inner workings of the Binder framework and the components of Android were of particular interest to us. We begin this report with a bit of history to set context, follow that with the Android software stack and then the crucial part about Binder.

2. HISTORY

Android was purchased by Google in 2005 from a company that goes by the same name as the OS. Android is the flagship software of the Open Handset Alliance (OHA), established by Google with members who are chip manufacturers, mobile handset makers, application developers or mobile carriers.

Binder interestingly was an idea originally conceived as OpenBinder to address issues encountered while developing the BeIA by Be Inc. and Palm OS Cobalt by Palm Inc. [2]. The

engineers at Google have then rewritten both the user space and driver code to better-fit with Linux centric design of Android [3].

3. ANDROID SOFTWARE STACK

The Software Stack or the architecture is four tiered.

3.1 APPLICATION LAYER

At the top is the Application layer, exclusively written in Java. The default applications like Phone, Contacts, Gallery, Calendar that ship with Android and those that users can download from Google PlayStore are all part of this layer. These Java applications execute on the Dalvik Virtual Machine (DVM), a virtual machine much like JVM but for Android.

Application Anatomy

1. **Activity**
   An activity is a single screen of the application that the user sees at one time, for e.g., the Contacts screen. However, the activity does not have to paint the entire screen. It displays the UI component and responds to a user or system initiated action. Activity code runs

only when the Activity GUI is visible and has focus. An application comprises of many such activities and Android maintains a history stack, 'back stack' of all the activities which are spawned. Maintaining this history enables Android to put out the activity on the screen quickly when a user returns to a previous displayed screen.

2. **Services**
   Service is an application component to enable an application to perform long running tasks in the background. This code runs when user interaction is needed, for e.g., applications continue to play music when you have navigated away to the home screen.

3. **Content Provider**
   The data storage and retrieval in Android applications is done via content providers. It is implemented as a subclass of *ContentProvider* that is discussed in the Application Framework section.

4. **Broadcast Receiver**
   A broadcast receiver is a component that responds to system-wide broadcast announcements. This part of the application decides what the application has to do, for instance, when the battery is low. The battery low message is broadcast.
   A broadcast receiver is implemented as a subclass of *BroadcastReceiver*

3.2 APPLICATION FRAMEWORK

The second layer is the Application framework layer implemented in Java which offers a repertoire of APIs catering to application developers. This layer exposes the operating system's capability in the form of services for effective and innovative application development. The components like Activity Manager, Resource manager etc., are through which applications interact with OS and other applications. [5]

1. **Content Providers**
   a primary building block providing encapsulated content to applications from a central repository of data. Content providers are necessary when the content or data is to be shared across applications. The content provider works along with ContentResolver to provide this data. The ContentResolver object in the client application's process and the ContentProvider object in the application that owns the provider automatically handle inter-process communication.
   public abstract class ContentProvider

2. **Activity Manager**
   The Activity Manager essentially handles the activity life cycle of an application. For example, it is responsible for creating the activity when an application starts, move an activity off screen when use navigates to a different screen. It interacts with all activities running in the system.
   public class ActivityManager

3. **Window Manager**
   The window manager is responsible for organizing the screen. It decides which application goes where and layering on the screen.

4. **Package Manager**
   It is a class for retrieving various kind of information related to the application packages that are currently installed on the device.
   public abstract class PackageManager

3.3 LIBRARIES

Android includes C / C++ native libraries which are exposed via the Application framework. So, these are called through Java interfaces. By native libraries, we mean code that is hardware specific. Most of these libraries are open source libraries which have been used without any changes. [6]

- The Bionic library is an exception; it is the optimized version of the Standard C library with the Apache license, like rest of Android, instead of GPL license of the GNU libc. The GNU libc has been rewritten taking into consideration the energy constraint of mobile devices that run Android.
- WebKit: Layout engine software designed to allow web browsers to render web pages. WebKit also powers the Apple Safari and Google Chrome browsers.
- Media Framework: the libraries support playback and recording of many popular audio and video formats including MPEG4, H.264, MP3, AAC, AMR, JPG, and PNG
- SQLite: A powerful yet light-weight SQL database engine. The applications use this to store and retrieve data.
- OpenGL: graphics libraries, OpenGL ES is a subset of OpenGL meant for Embedded Systems

- SGL: acronym for Scalable Graphics Libraries, renders 2D graphics

- OpenSSL: the open source tool kit that implements Secure Socket Layer (SSL)

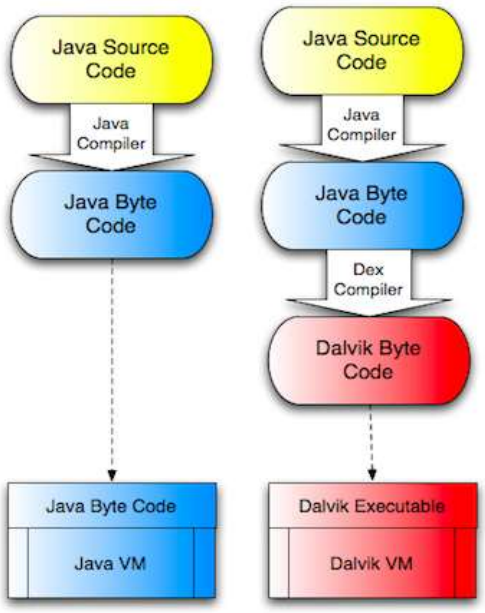
3.4 ANDROID RUNTIME

3.4.1 Core Libraries

Android re-implements core Java libraries that the layers above use. Android includes most of Java 5 Standard Edition functionalities in its implementation.

3.4.2 Dalvik Virtual Machine (DVM)

Android does not use the standard Java Virtual Machine but uses the DVM which has been tailor made for mobile devices and is an integral part of Android. DVM ensures that applications can run in systems with relatively smaller RAM, slower processors and without swap space in comparison to desktops. This is especially relevant as the mobile devices are battery powered and memory efficiency of programs is very important. As a consequence, Dalvik itself is small and operates with tighter overhead constraints. DVM is a register-based architecture unlike JVM which is stack-based. Dalvik has its own byte code and hence the standard Java compiler generated byte code has to be converted to a .dex file which DVM can execute. The .dex is the DVM compatible byte code which is compact and efficient. The uncompressed .dex file is smaller than a compressed .jar file! [6]. All applications run as separate processes with their own instance of the DVM. This adds to the security of Android.



3.5 KERNEL

Android uses the Linux kernel, which is also open source, for device drivers, memory management, process management, file system and network management. The kernel has some features added, some dropped from the Linux kernel. Android does not support Sys V IPC mechanisms but instead uses IPC Binder. The binder driver that is used for IPC is included to the kernel along with low memory killer, power management, logger and ashmem (Anonymous SHared MEMory)[7]. There have been attempts to merge some of these modules to the Linux kernel mainline with limited success. As of now, Binder, low memory killer, Logger and Pmem have already made it to the linux-next staging repository.

- **Low Memory Killer**- Kills processes as available RAM becomes low. The lowmemorykiller driver lets user-space specify a set of memory thresholds where processes within a range can be killed when the available memory reduces. This is necessary as when an application is closed in Android, the process does not get killed but instead stays in memory. This reduces the start-up time when the application has to be used again. This driver is built on top of the Linux
- **Logger** – system logging facility
- **ashmem** – is mechanism to share blocks of data across processes, similar to POSIX shm. The modifications to POSIX shm help Android to reclaim memory blocks not currently in use.
- **Power Management** – Android has to operate with substantially limited energy footprint. Consequentially, Android implements a power management driver on top of standard Linux power management. Applications use user space libraries to inform Power Management framework of their constraints.
- **pmem** – allocates physically contiguous memory

4. BINDER

4.1 INTRODUCTION

The architectural paradigm of android applications is based on a distributed component model. The various components of an android application are decoupled from one another for modularity and scalability. These components can be part of the same process or different processes. If these components reside in different processes, then the components have to communicate with one another using the underlying Inter-process mechanisms exposed by the android platform. But these distributed components have to be present on the same host, because the IPC mechanism supported by the android platform does not allow communication across hosts.

It is not uncommon on an android platform to have multiple distributed components executing as disparate processes and coordinating with each other to perform a task. In fact it would not be wrong to say that IPC is ubiquitous throughout the android platform. To justify this assertion, let us consider an mp3 player application. An mp3 player usually runs as 2 processes, one processes is the Activity, which is the UI of the application, the other process is the Service, which runs in the background and runs the business logic of the entire application.

4.1.1 IPC mechanisms

Since the android platform runs a Linux kernel underneath, it supports all the traditional IPC mechanism supported by a Linux kernel. Here is a brief overview of the traditional IPC mechanism.

a) ***Pipes***: Pipes are unidirectional byte-streams that connect the standard output from one process with the standard input of another process.

b) ***Message Queues***: maintains a queue of messages to which processes can read to and write from, thereby achieving IPC.

c) ***Shared Memory***: A common memory location which is accessible by all communicating processes. IPC is achieved by writing to and reading from the shared memory location.

d) ***Semaphores***: A semaphore is a shared variable on which processes can signal and wait thereby achieving IPC.

e) ***Signals***: A process can send signals to processes with the same uid and gid or in the same process group.

f) ***Sockets***: Sockets are bidirectional communication streams. Two processes can communicate with byte-streams by opening the same socket.

4.1.2 Android IPC

Although android supports all of the traditional IPC mechanisms supported by the Linux kernel, these mechanisms are not used by OS libraries and platform API's to perform IPC, instead the android platform uses an elaborate framework which spans over the Linux kernel layer, the middle-ware and the application layer. This framework is called the Binder framework.

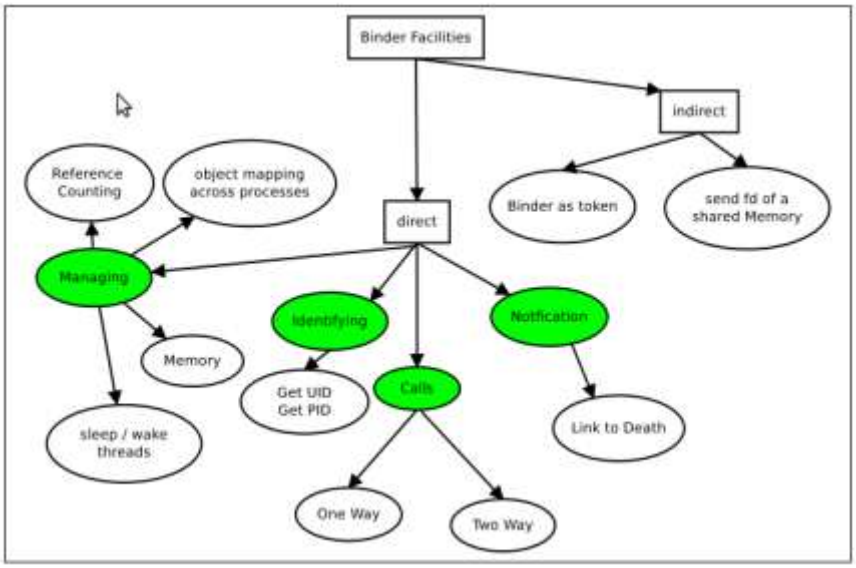
Binder: is defined as a low overhead Remote Procedure call utility which facilitates synchronous reliable communication across processes. It is an elaborate framework and a binder kernel driver resides at the heart of this framework.

4.2 BINDER FRAMEWORK

4.2.1 Introduction

The Binder has the facility to provide bindings to functions and data from one execution environment to another. Binder in Android is a customized implementation of OpenBinder. The core concepts remained the same, but many details have changed over the newer releases of android. The original OpenBinder code is no longer being developed [8] .

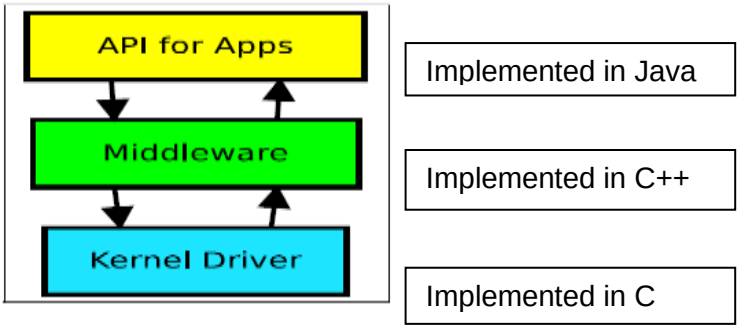
4.2.2 Binder Facilities



The above figure shows the various facilities supported by the binder in addition to merely supporting IPC mechanisms. By using the binder directly a process can avail the following facilities, when a client wants to communicate with a server over the binder, on successfully establishing the communication, the client receives a proxy object to the server. Using this object the binder can maintain object mapping and object referencing (i. e which object belongs to what process). The binder can provide notifications to clients about presence of a server or service. The binder also provides functionality to wake up sleeping client or server threads on arrival of notifications, requests and acknowledge packets. Indirectly the binder could be used as a security token, to restrict access of certain services to certain clients. It can also be used to transfer file descriptors across processes.[1]

4.2.3 IPC internals from Top down

The Binder framework is implemented over different layers of abstraction. All Java applications access the binder's functionality through the Binder interface exposed at the Java API layer. The Java implementation of the Binder interface in turn access the Binder interface implemented at the middle-ware layer. The middle-ware is exposed to the Java layer through JNI wrappers. The middle ware layer is actually responsible for performing marshalling and unmarshalling of objects and communicating with the Binder kernel driver using a low level communication protocol.



The abstraction layers ensures that the application developers do not have to be burden themselves with the lower level implementation details and the generic abstraction layer caters to all applications which intend to perform IPC.

4.2.4 IPC over the Application Layers

Applications perform IPC through the IBinder interface. The service provides an IBinder object for the client. The client uses this object directly call the remote methods implemented by the service.

The service has to implement the *onBind()* callback method to return the interface for the client.

```
public IBinder onBind(Intent intent)
{
return mBinder; // Return the interface
}
```

The client calls the *bindService* method and implements the *onServiceConnected()* callback. The client gets the object that service gives within the *onServiceConnected()* method.

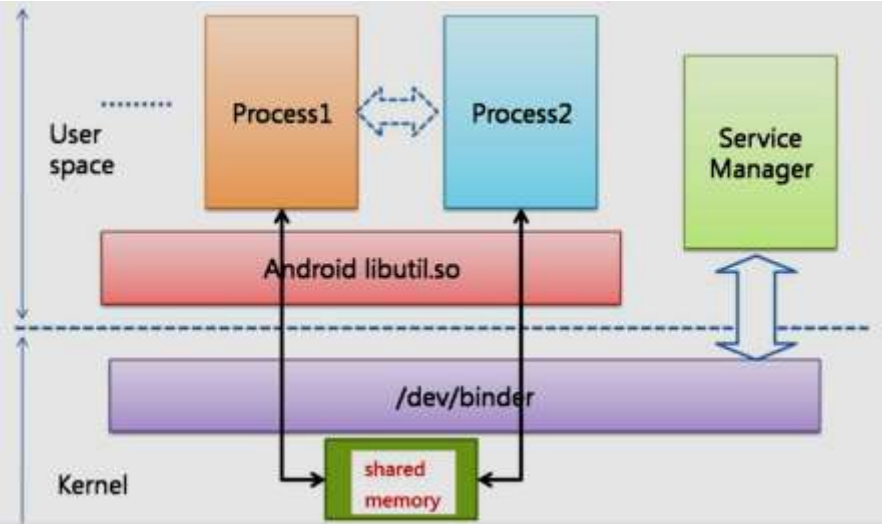
```
public void onServiceConnected(ComponentName className, IBinder service)
 {
mIRemoteService = RemoteService.Stub.asInterface(service);
 }
```

The IBinder API *transact()* which is matched by *Binder.onTransact()* method provide IPC mechanism through synchronous method calls(blocking). The transact method sends data in the form of a *parcel*,a generic data buffer that also maintains metadata about the objects that are transferred.

4.2.5 IPC over the middleware

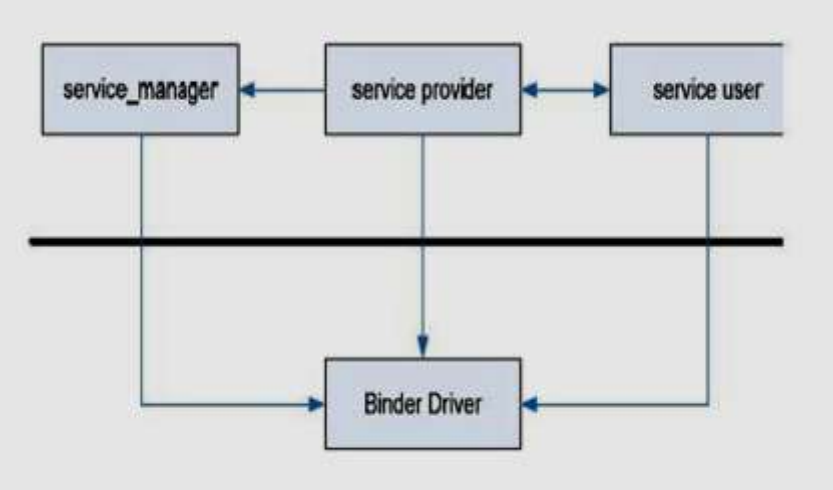
The middle-ware is implemented in C++ and the functionality of the middle-ware is exposed to application layer through JNI wrappers. This is a high level interface responsible for marshaling and un-marshaling packets, creating and managing thread pools and providing reliability to the application layers. This framework accesses the kernel driver for the application and implements the communication protocol. The Binder middle-ware functionalities are part of a shared library libutil.so. This shared library is built mainly from these source files [8]

• frameworks/base/include/utils/IInterface.h
• frameworks/base/include/utils/Binder.h
• frameworks/base/include/utils/BpBinder.h
• frameworks/base/include/utils/IBinder.h
• frameworks/base/include/utils/Parcel.h
• frameworks/base/include/utils/IPCThreadState.h
• frameworks/base/include/utils/ProcessState.h
• frameworks/base/libs/utils/Binder.cpp
• frameworks/base/libs/utils/BpBinder.cpp
• frameworks/base/libs/utils/IInterface.cpp
• frameworks/base/libs/utils/ProcessSTate.cpp
• frameworks/base/libs/utils/IPCThreadState.cpp



4.2.5.1 Service Manager

How does a process receive the handle to a target process? Although this is intelligently abstracted at the application layer, it serves as an important piece in the overall framework and is implemented at the middle-ware layer.



Service manager provide service manager service to other process. It must be started before any other service gets running. It first open "/dev/binder" driver and then call BINDER_SET_CONTEXT_MGR ioctl to let binder kernel driver know it acts as a manager. service manager runs first, it will register itself with a handle 0. The other process must use this handle to talk with service manager. Using 0 as the handle, service provider registers a service with the service manager. So it will generate a handle (assume 10) for the service. Service manager will store the name and handle. Using 0 as the handle, the client tries to get a particular service, service manager on finding that particular service will also return a handle of the server, so that the client can communicate with the server directly. The class hierarchy of the Binder framework at the middleware layer is as shown below

Important methods which resolve to the Binder kernel driver are:[2]

IPCState::transact(): The transact method is responsible for sending data to the target process. The calls from the application layer resolve to the transact method which will eventually invoke ioctl() system call with approriate arguments.

IServiceManager::addService(): This method is responsible for regstering a service exposed by the server to the service manager.

IServiceManager::getService(): This method is responsible for retrieving a particulat service requested by a client. It returns an IBinder object which is a proxy of the server object.

At the target server side, the binder wakes up the BC_THREAD_LOOPER thread. From here the ontransact() method is called upto the application layer, on the way up the call stack marshalling of raw data is performed and it finally reaches the application layer.

4.2.6 IPC at the Kernel Level

The Binder kernel driver is the crux of the Binder framework. The kernel driver is a kernel module written in C. The kernel module is built from the source files:[8]

- /drivers/staging/android/binder.c
- /drivers/staging/android/binder.h

The Binder kernel driver supports the file operations open, mmap, release, poll and the system call ioctl. The higher layers accesses the Binder driver through these file operations. The device file associated with the binder kernel module is "/dev/Binder". The open system call establishes a connection to the Binder driver and assigns it a file descriptor, and the release operation closes the connection. The mmap operation is needed to map Binder memory. The most significant operation is the ioctl system call. The higher layers send and receive all messages through the ioctl() system call. All interactions transpire through a small set of ioctl commands. These commands are: [8]

BINDER WRITE READ is the most important command; it submits a series of transmission data.

BINDER SET MAX THREADS sets the number of maximal threads per process to work on requests.

BINDER SET CONTEXT MGR sets the context manager. It can be set only one time successfully and follows the first come first serve pattern.

BINDER THREAD EXIT This command is sent by middleware, if a binder thread exits

BINDER VERSION returns the Binder version number

To initiate an IPC transaction, ioctl call with BINDER_READ_WRITE command is invoked The data to be passed to the ioctl() call is of the type struct binder_write_read.[3]
ioctl(fd, BINDER_WRITE_READ, &bwt);

```
struct binder_write_read
{
        ssize_t write_size;             /*bytes to write*/
    ssize_t write_consumed;  /*bytes consumed*/
        const void* write_buffer;
    ssize_t read_size;          /*bytes to be read*/
        void* read_buffer;              /*bytes consumed*/
};
```

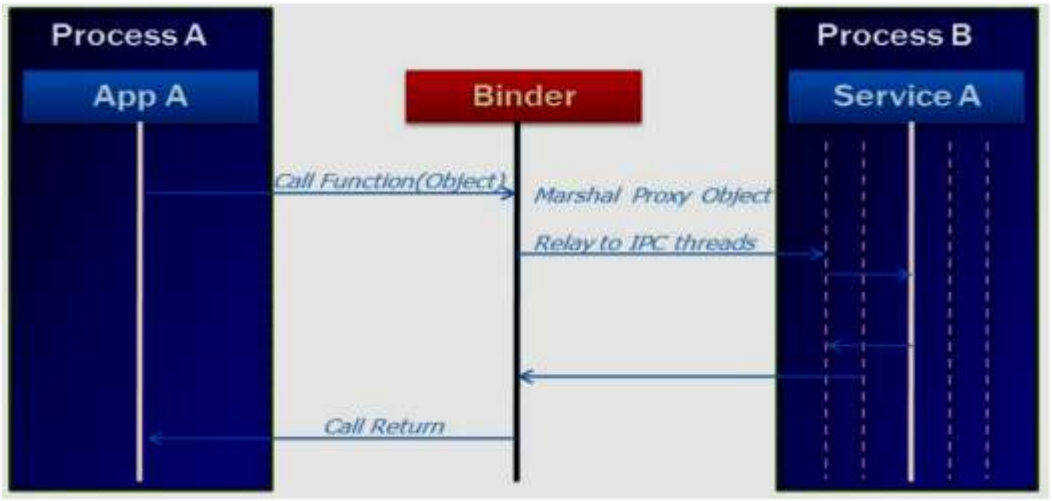
The write buffer contains an *enum* BC_TRANSACTION followed by a *binder_transaction_data*. In this structure target is the handle of the object that should receive the transaction. The code refers to the Method ID. [3]

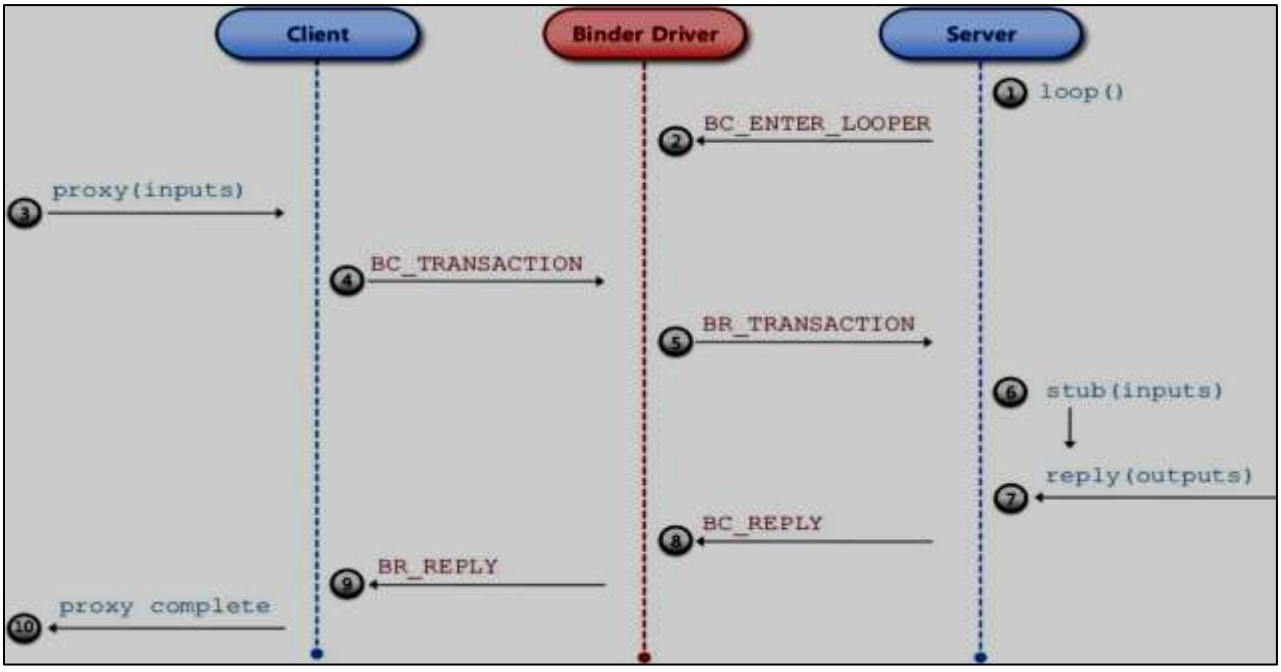
```
struct binder_transaction_data {
   union {
      size_t  handle; /* target descriptor of command transaction */
      void    *ptr;   /* target descriptor of return transaction */
   }target;
   void *cookie;    /* target object cookie */
   unsigned int code;      /* transaction command */
   /* General information about the transaction. */
   unsigned int    flags;
   pid_t       sender_pid;
   uid_t       sender_euid;
   size_t      data_size;  /* number of bytes of data */
   size_t      offsets_size;   /* number of bytes of offsets */
   .....
};
```

The kernel driver does not start threads to dispatch requests to a target server, it is the responsibility of the server to main a pool of threads and listen to the incoming requests by entering a polling loop. The target server should execute these commands BC REGISTER LOOPER, BC ENTER LOOPER and BC EXIT LOOPER to notify the binder about the looping threads.

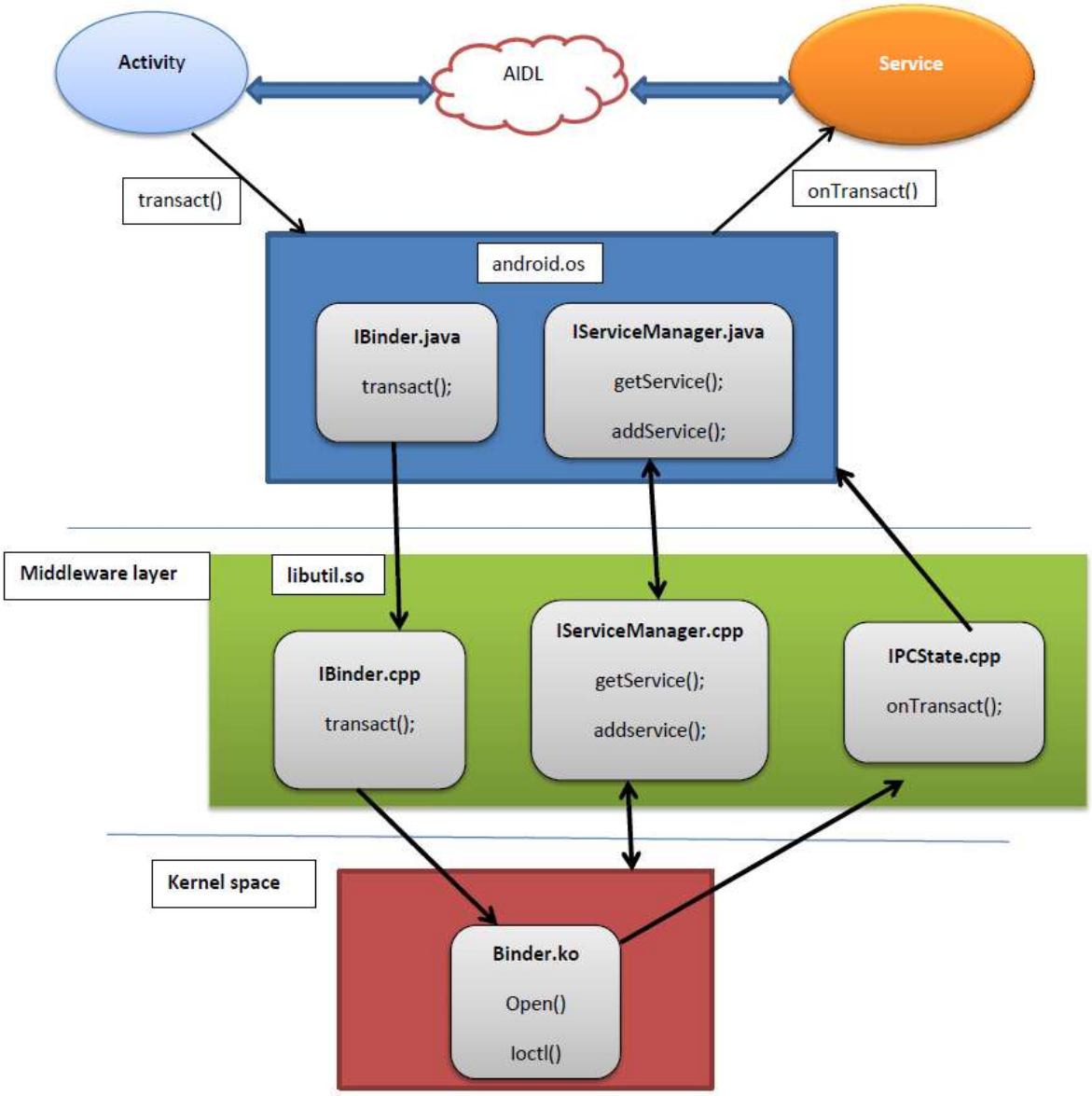
4.2.6.1 Binder Transactions

The BC REPLY command is used by the server to reply to a received BC TRANSACTION. The Binder driver takes care of delivering the reply to the waiting thread. The Binder driver copies the transmission data from the user memory address space of the sending process to its kernel space and then copies the transmission data to the destination process. This is achieved by the copy from user and copy to user command of the Linux kernel.[8][9]





4.3 HOW IT ALL FALLS IN PLACE



4.4 WHY BINDER OVER TRADITIONAL IPC

Binder has additional features that sockets don't have. For example: binder allows passing file descriptors across processes. Pipes cannot perform RPC, Object reference counting, object mapping. Binder has elaborate data referencing policies; it is not a simplistic kernel driver.

4.5 LIMITATIONS OF BINDER

Binders are used to communicate over process boundaries since different processes don't share a common VM context - no more direct access to each other's 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.[9]

4.6 EVALUATING BINDER'S PERFORMANCE OVER TRADITIONAL IPC MECHANISMS

Goal

To evaluate the performance of a native application using binder IPC against a native application that uses traditional IPC.

Experimental Setup

The target device chosen to carry out the experiment is an Android emulator with X86 architecture and android 4.1 as the operating system version. Native binaries are compiled for this target architecture using android NDK tool chain executables. The binaries obtained are pushed onto the target emulator device using the adb shell. Since the device is not rooted, the executable is moved onto /data/local/tmp of the target device. Since perf (Linux profiling tool) is already integrated as part of the OS deliverable, there is no need to install it separately.

Experiment

For each of the executable we run the following commands
perf stat ./pipes_impl //A program using pipes
perf stat ./sockets_impl //A program using sockets
perf stat ./client_impl //A program using the Binder

Results

perf stat ./pipes_impl
Performance counter stats for './pipes_impl':
<not counted> task-clock
<not counted> context-switches
<not counted> CPU-migrations
<not counted> page-faults
<not counted> cycles
<not counted> stalled-cycles-frontend
<not counted> stalled-cycles-backend
<not counted> instructions
<not counted> branches
<not counted> branch-misses
0.072900029 seconds time elapsed

perf stat ./sockets_impl
Performance counter stats for './sockets_impl':
<not counted> task-clock
<not counted> context-switches
<not counted> CPU-migrations
<not counted> page-faults
<not counted> cycles
<not counted> stalled-cycles-frontend
<not counted> stalled-cycles-backend
<not counted> instructions
<not counted> branches
<not counted> branch-misses
2.492557716 seconds time elapsed

---

perf stat ./client_impl
Performance counter stats for './client_impl':
<not counted> task-clock
<not counted> context-switches
<not counted> CPU-migrations
<not counted> page-faults
<not counted> cycles
<not counted> stalled-cycles-frontend
<not counted> stalled-cycles-backend
<not counted> instructions
<not counted> branches
<not counted> branch-misses
1.163930349 seconds time elapsed

Result Analysis

The results show that the performance stats has not been counted for any of the three cases, the failure could be attributed to the fact that, since *perf* accesses Hardware counters of a device to collect stats and since this experiment is being performed on an emulator which does not have hardware counters, the results are all null. The possibility of the emulator accessing the host PC's hardware counters to obtain results is far-fetched and would be incorrect because the host PC would not collect stats local to one particular application running on the emulator.

5 REFERENCES

[1] http://en.wikipedia.org/wiki/Android_%28operating_system%29

[2] http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html

[3] http://elinux.org/Android_Binder

[4] http://developer.android.com/reference/

[5] http://developer.android.com/guide/

[6] http://ofps.oreilly.com/titles/9781449390501/The_Stack.html

[7]http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Porting-Android-to-a-new-device/

[8]http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf

[9] http://0xlab.org/~jserv/android-binder-ipc.pdf

[10] http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderIPCMechanism.html