Android Architecture and Binder

Dhinakaran Pandiyan
Saketh Paranjape
Android Software stack
Anatomy of an Android application

• Activity
  UI component typically corresponding of one screen

• Service
  Background process without UI (E.g mp3 player).

• Content Manager
  Enables applications to share data (E.g Contacts shared b/w all applications)

• Broadcast Receiver
  Responds to external events, can wake up your process (E.g SMS, Phone ring, Low battery)
Anatomy of an Android Application
IPC is ubiquitous throughout the Android platform

• Example: An mp3 player application
• It executes as 2 separate process

mp3 player UI [Activity]

mp3 player backend [Service]
IPC mechanism in android

- In GNU/Linux
  Pipes
  Shared Memory
  Message Queue

- In Android
  Binder
Why Binder over conventional IPC

• Binder has additional features that sockets don't have. E.g., 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.
Binder and Open Binder

• Developed under the name OpenBinder by Palm Inc.
• Android Binder is the customized re-implementation of OpenBinder.
Binder

- A kernel driver to facilitate inter-process communication
- Lightweight RPC (Remote Procedure Communication) mechanism
- Per-process thread pool for processing requests
- Synchronous communication b/w processes
IPC internals from Bottom Up

- IPC over Binder kernel driver
- IPC over the middleware
- IPC over the Application layer
• Binder Driver supports the file operations open, mmap, release, poll and the system call ioctl.

• The first thing an application must do is open the Binder kernel module("/dev/Binder").

• This associates a file descriptor with that thread.

• The kernel module uses the descriptor to identify the initiators and recipients of Binder IPCs.
• All interactions with the driver will happen through a small set of ioctl() commands.
  - BINDER_WRITE_READ
  - BINDER_SET_MAX_THREADS
  - BINDER_SET_CONTEXT_MGR
  - BINDER_THREAD_THREAD_EXIT
  - BINDER_VERSION

• The key command is BINDER_WRITE_READ, which is the basis for all IPC operations.

  ```
  ioctl(fd, BINDER_WRITE_READ, &bwt);
  ```

• To initiate an IPC transaction, ioctl call with BINDER_READ_WRITE_WRITE command is invoked
The data to be passed to the ioctl() call is of the type struct binder_write_read

```
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 bcTRANSACTION 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.

```c
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 */
    ........
};
```
Send data from A to B

```
struct binder_transaction_data {
    size_t handle;
    unsigned int code;
    void *buffer;
};
```

```
switch (binderProtocol) {
    case BC_TRANSACTION: {...}
    case BR_TRANSACTION: {...}
}
```
How does a user process receive the handle to a target process?
• Binder performs mapping of objects between two processes.
• A pool of threads is associated with each service to process incoming IPC
• Service_manager provides registry service to other processes
• It must be started before any other service gets running
• It first opens “/dev/binder” driver
• It then calls 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
• The binder 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 the handle '10' of the server, so that the client can communicate with the server directly
IPC over middleware

- C++ framework over the binder kernel driver
- A high level interface responsible for marshalling and unmarshalling
- Access the kernel driver for the application
Native Binder framework
public interface **IBinder**

boolean transact(int code, Parcel data, Parcel reply, int flags)

public class **Binder** implements **IBinder**

onTransact(int code, Parcel data, Parcel reply, int flags)
Bound Service

• Return an IBinder for the client to use
• Implement onBind() and other methods for RPC
• Create a .aidl file which generates interface file in java

mBinder = new IRemoteService.Stub()
public IBinder onBind(Intent intent)
{
    // Return the interface
    return mBinder;
}
Client

• Calls `bindService()` to bind to the service
• Implements `onServiceConnected()` callback

```java
public void onServiceConnected(ComponentName className, IBinder service)
{
    mIRemoteService = RemoteService.Stub.asInterface(service);
}
```
What to do with IPC?

• Passing Objects over IPC
  send parcels
  transact() and onTransact()

• Calling an IPC Method
  Use IBinder object at the client to call methods implemented by the server
  mlIRemoteService.getService()
Parcel

Delivering arguments of method

"flatten"

transmit

"unflatten"
IPC overview across layers

- **Activity**
- **AIDL**
- **Service**
  - `onTransact()`

**Middleware layer**
- **libutil.so**
  - `IBinder.cpp`
    - `transact()`
  - `IServiceManager.cpp`
    - `getService()`
    - `addService()`
  - `IPCState.cpp`
    - `onTransact()`

**Kernel space**
- **Binder.ko**
  - `Open()`
  - `ioctl()`
Other Android specific features...

- **DVM: A JVM for Android**
  - Compact byte code
  - Register based implementation
  - Native Libraries bypass DVM

- **Power Management**
  - Battery Power – Android tries to put device to sleep
  - A mechanism to say that your app wants device to stay on i.e., don’t sleep
    - `acquire(long timeout)`
    - `void release()`
Project plan...

- Analyze binder performance
- Trace Binder IPC data/control flow

References:

http://www.cubrid.org/blog/dev-platform/binder-communication-mechanism-of-android-processes/
http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf
http://developer.android.com/guide/
http://app-solut.com/
http://mindtherobot.com
http://www.slideshare.net/jserv/android-ipc-mechanism by Jim Huang
Analysis of the Android Architecture - Stefan Brähler
http://elinux.org/Android_Binder

Service Manager Run [blogimg.chinaunix.net/blog/upfile2/081203105044.pdf]