```
$mmm frameworks/base/camera/tests/CameraServiceTest/
$adb push out/target/product/generic/system/bin/CameraServiceTest  /system/bin
$mmm frameworks/base/libs/camera/
$adb push out/target/product/generic/system/lib/libcamera_client.so /system/lib
```

I will use call IAudioFlinger::setMode API as a sample scenario to show how the Android IPC system works. AudioFlinger is a service in the media_server program

```
# service list
# rpcperftest SurfaceFlinger

#mmm packages/experimental/RpcPerformance/
adb push out/target/product/smdkc110/system/xbin/rpcperftest /system/xbin/
```

## Service Manager Run

service_manager provide service manager service to other process. It must be started before any other service gets running.

```
int main(int argc, char **argv)
{
        struct binder_state *bs;
        void *svcmgr = BINDER_SERVICE_MANAGER;

        bs = binder_open(128*1024);

        if (binder_become_context_manager(bs)) {
                LOGE("cannot become conte
                return -1;
        }

        svcmgr_handle = svcmgr;
        binder_loop(bs, svcmgr_handler);
        return 0;
}
```

code location:
frameworks/base/cmds/servicemanager

init.rc

```
service servicemanager /system/bin/servicemanager
    user system
    critical
    onrestart restart zygote
    onrestart restart media
```

Service Manager作为一个Server大总管，本身也是一个server。既然是一个server就要时刻准备为客户端提供服务。Service Manager调用binder_loop进入到循环状态，并提供了一个回调函数，等待用户的请求。注意他的Service Manager的客户端既包括应用程序（查询和获取服务），也包括Server（注册服务）。
Service Manager的客户怎样请求其服务呢？客户需要在自己进程中创建一个服务器代理。现在没有地方去查询服务，那么客户怎样生成代理对象呢？答案是binder设备（/devbinder）为每一个服务维护一个句柄，调用binder_become_context_manager函数变为“ Server大总管 ”的服务，他的句柄永远是0，是一个“ 众所周知 ”的句柄，这样每个都可以通过binder机制在自己的进程空间中创建一个Service Manager代理对象了。其他的服务在binder设备在设备中的句柄是不定的，需要向“ Server大总管 ”查询才能知道

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. Then it enters into a loop to wait for any data from other process.

binder_become_context_manager的
代码在frameworks/base/cmds/servicemanager/binder.c
代码就是一句：
ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
frameworks/base/libs/binder/ProcessState.cpp
ProcessState::becomeContextManager也调用了这个ioctl

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
        int res;
        struct binder_write_read bwr;
        unsigned readbuf[32];

        bwr.write_size = 0;
        bwr.write_consumed = 0;
        bwr.write_buffer = 0;

        readbuf[0] = BC_ENTER_LOOPER;
        binder_write(bs, readbuf, sizeof(unsigned));

        for (;;) {
                bwr.read_size = sizeof(readbuf);
                bwr.read_consumed = 0;
                bwr.read_buffer = (unsigned) readbuf;
```

```
            res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);

            if (res < 0) {
                LOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
                break;
            }

            res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
            if (res == 0) {
                LOGE("binder_loop: unexpected reply?!\n");
                break;
            }
            if (res < 0) {
                LOGE("binder_loop: io error %d %s\n", res, strerror(errno));
                break;
            }
        }
}
```

Pay attention to BINDER_SERVICE_MANAGER.

```
/* the one magic object */
#define BINDER_SERVICE_MANAGER ((void*) 0)
```

BINDER_SERVICE_MANAGER is the registered handle for service_manager. The other process must use this handle to talk with service_manager.

## Get IServiceManager

To get an IServiceManager instance the only method is to call defaultServiceManager implemented in IServiceManager.cpp.

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }

    return gDefaultServiceManager;
}
```

service manager存在是因为客户进程与服务进程不相识，不像socket用于父子进程通讯，父进程认识子进程，两个进程自己就可以建立IPC（父进程把socket句柄传给子进程）。 Binder用于更广泛的进程通信， service manager是大家公认的中介（媒人），也就是不管客户进程，还是服务进程，都能直接获取的，所以，各种客户和服务都要包含库libbinder，都需要使用defaultServiceManager，而不管使用何种接口；从另一个方面，只要接口不变，服务的实现可以有不同进程提供，就像BREW等使用的COM, 客户与服务隔离开来了，弱耦合.

service_manager 是个进程， 而IServiceManager是共享库libbinder的一个接口。 从 服务角度来看，service_manager 提供了一个服务， 而凡是使用了libbinder库的进程都是他的客户， 这些客户又可以分为客户进程和服务进程。ServiceManager 提供的服务有get service， add service，list services等等， frameworks/base/includes/binder/IServiceManager.h定义了这些方法， 如 GET_SERVICE_TRANSACTION = IBinder:: FIRST_CALL_TRANSACTION， GET_SERVICE_TRANSACTION 不是保留的方法ID,而是正常的方法ID. 事实， 每个接口的方法ID都是以FIRST_CALL_TRANSACTION开始 （ frameworks/base/includes/binder/IBinder.h）
如./media/libmedia/IAudioFlinger.cpp:   CREATE_TRACK = IBinder::FIRST_CALL_TRANSACTION,
cmds/servicemanager/binder.h, libs/binder/IServiceManager.cpp 都说明 “ "android.os.IServiceManager",这个接口。
服务进程向service manager注册， 客户进程向service manager 查询， 最后， 客户进程和服务进程建立联系， 这是一个三角关系

it will first get a ProcessState instance through ProcessState::self(). One process has only one ProcessState instance. ProcessState will open "/dev/binder" driver for IPCThreadState use.

```
ProcessState::ProcessState()
    : mDriverFD(open_driver())
```

Now we have an instance of ProcessState, let's look at the getContextObject.

```
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
{
    if (supportsProcesses()) {
        return getStrongProxyForHandle(0);
    } else {
        return getContextObject(String16("default"), caller);
    }
}
```

The board support binder driver, so we will get into getStrongProxyForHandle. (Handle 0 is reserved for service manager, which will be explained later.)

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;

    AutoMutex _l(mLock);

    handle_entry* e = lookupHandleLocked(handle);

    if (e != NULL) {
        // We need to create a new BpBinder if there isn't currently one, OR we
        // are unable to acquire a weak reference on this current one.   See comment
        // in getWeakProxyForHandle() for more info about this.
        IBinder* b = e->binder;
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
            b = new BpBinder(handle);
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        } else {
            // This little bit of nastyness is to allow us to add a primary
            // reference to the remote proxy when this team doesn't have one
            // but another team is sending the handle to us.
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }
```

```
        return result;
}
```

The first time b will be NULL, so the code will new an BpBinder instance. BpBinder is a base proxy class for remote binder object.

```
BpBinder::BpBinder(int32_t handle)
        : mHandle(handle)
        , mAlive(1)
        , mObitsSent(0)
        , mObituaries(NULL)
{
        LOGV("Creating BpBinder %p handle %d\n", this, mHandle);

        extendObjectLifetime(OBJECT_LIFETIME_WEAK);
        IPCThreadState::self()->incWeakHandle(handle);
}
```

IPCThreadState::incWeakHandle will add a BC_INCREFS command in output buffer.

```
void IPCThreadState::incWeakHandle(int32_t handle)
{
        LOG_REMOTEREFS("IPCThreadState::incWeakHandle(%d)\n", handle);
        mOut.writeInt32(BC_INCREFS);
        mOut.writeInt32(handle);
}
```

Now getContextObject returns a BpBinder instance, it will be interface_cast to IServiceManager. interface_cast is defined in IInterface.h. It will be extended as:

```
inline sp<IServiceManager> interface_cast(const sp<IBinder>& obj)
{
        return IServiceManager::asInterface(obj);
}
```

Now let’s take a look at definition of IServiceManager.

```
class IServiceManager : public IInterface
{
public:
        DECLARE_META_INTERFACE(ServiceManager);

        /**
          * Retrieve an existing service, blocking for a few seconds
          * if it doesn't yet exist.
          */
        virtual sp<IBinder>                 getService( const String16& name) const = 0;

        /**
          * Retrieve an existing service, non-blocking.
          */
        virtual sp<IBinder>                 checkService( const String16& name) const = 0;
```

```
    /**
      * Register a service.
      */
    virtual status_t            addService( const String16& name,
                                            const sp<IBinder>& service) = 0;

    /**
      * Return list of all existing services.
      */
    virtual Vector<String16>    listServices() = 0;

    enum {
        GET_SERVICE_TRANSACTION = IBinder::FIRST_CALL_TRANSACTION,
        CHECK_SERVICE_TRANSACTION,
        ADD_SERVICE_TRANSACTION,
        LIST_SERVICES_TRANSACTION,
    };
};
```

DECLARE_META_INTERFACE macro is defined as follows in IInterface.h. And it will be extended to the following code:

```
    static const String16 descriptor;
    static sp<IServiceManager> asInterface(const sp<IBinder>& obj);
    virtual String16 getInterfaceDescriptor() const;
```

As you can see, DECLARE_META_INTERFACE macro declares two functions which are implemented by IMPLEMENT_META_INTERFACE macro in IServiceManager.cpp.

```
IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager");
```

The code will be extended like this.

```
    const String16 IServiceManager::descriptor(NAME);
    String16 IServiceManager::getInterfaceDescriptor() const {
        return IServiceManager::descriptor;
    }
    sp<IServiceManager> IServiceManager::asInterface(const sp<IBinder>& obj)
    {
        sp<IServiceManager> intr;
        if (obj != NULL) {
            intr = static_cast<IServiceManager*>(
                obj->queryLocalInterface(
                        IServiceManager::descriptor).get());
            if (intr == NULL) {
                intr = new BpServiceManager(obj);
            }
        }
        return intr;
        }
```

So IServiceManager::asInterface will finally new a BpServiceManager instance and return it to user. BpServiceManager works as a proxy for remote BnServiceManager. Any operation on IServiceManager now actually is to call the corresponding virtual functions in BpServiceManager.

Summary:
This section gives out details in how to get a proxy object for remote object.
Assume you want to implement your own service IFunnyTest, you must do the following:

- Put DECLARE_META_INTERFACE(FunnyTest) macro in your interface header file.
- Put IMPLEMENT_META_INTERFACE(Funnytest, “your unique name”) macro in your interface source file.
- Implement your own BpFunnyTest class.

## Generate AudioFlinger Service

media_server program will start the AudioFlinger service. Here is the code.

```
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();
        ProcessState::self()->startThreadPool();
        IPCThreadState::self()->joinThreadPool();
}
```

对照 Get IAudioFlinger
两者都需调用defaultServiceManager()
，　不同的是，这里要addService，　而IAudioFlinger客户需要getService。 当然，这里需要调用下面的函数，启动，进入循环，提供服务
ProcessState::self()->startThreadPool();
IPCThreadState::self()->joinThreadPool();

AudioFlinger will call IServiceManager::addService, which is a RPC call.

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

AudioFlinger inherits from BnAudioFlinger, which is a template BnInterface class.

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

BnInterface inherits from BBinder.

```
template<typename INTERFACE>
```

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

protected:
    virtual IBinder*            onAsBinder();
};
template<typename INTERFACE>
IBinder* BnInterface<INTERFACE>::onAsBinder()
{
    return this;
}
```

According to BnInterface’m implementation we know that the parameter passed to IServiceManager::addService is the address of new AudioFlinger instance. BBinder derives from IBinder, its transact function is to call virtual function onTransact.

```
status_t BBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    data.setDataPosition(0);

    status_t err = NO_ERROR;
    switch (code) {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
            err = onTransact(code, data, reply, flags)
            break;
    }

    if (reply != NULL) {
        reply->setDataPosition(0);
    }

    return err;
}
```

```
FIRST_CALL_TRANSACTION  = 0x00000001,
LAST_CALL_TRANSACTION   = 0x00ffffff,

PING_TRANSACTION      = B_PACK_CHARS
('_','P','N','G'),
DUMP_TRANSACTION      = B_PACK_CHARS
('_','D','M','P'),
INTERFACE_TRANSACTION   = B_PACK_CHARS('_',
'N', 'T', 'F'),


#define B_PACK_CHARS(c1, c2, c3, c4) \
  ((((c1)<<24)) | (((c2)<<16)) | (((c3)<<8)) | (c4))
所以，PING_TRANSACTION是保留的id。
```

The most important virtual functions is onTransact. BnAudioFlinger implemented the virtual function. For the scenario, we just need to focus on SET_MODE branch.

```
status_t BnAudioFlinger::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
          case SET_MODE: {
               CHECK_INTERFACE(IAudioFlinger, data, reply);
               int mode = data.readInt32();
               reply->writeInt32( setMode(mode) );
               return NO_ERROR;
          } break;
```

media_server will enter into a loop through IPCThreadState::joinThreadPool, just like service_manager, it will be waited data from other process in talkWithDriver.

```
void IPCThreadState::joinThreadPool(bool isMain)
{
     mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);

     status_t result;
     do {
          int32_t cmd;
          result = talkWithDriver();
          if (result >= NO_ERROR) {
               size_t IN = mIn.dataAvail();
               if (IN < sizeof(int32_t)) continue;
               cmd = mIn.readInt32();
               result = executeCommand(cmd);
          }

          // Let this thread exit the thread pool if it is no longer
          // needed and it is not the main process thread.
          if(result == TIMED_OUT && !isMain) {
               break;
          }
     } while (result != -ECONNREFUSED && result != -EBADF);

     mOut.writeInt32(BC_EXIT_LOOPER);
     talkWithDriver(false);
}
```

Assume you want to implement your own service IFunnyTest, you must do the following:

- Implement your own BnFunnyTest class.
- In the process your service running, call IPCThreadState::joinThreadPool to start the loop for binder.

## RPC Call IServiceManager::addService

We called IServiceManager::addService, which means call BpServiceManager::addService.

```
     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.readInt32() : err;
    }
```

Parcel is simple. We can think it as a continuous buffer. Pay attention here, service parameter points to BBinder object(AudioFlinger derived from Bn)

```
status_t Parcel::writeStrongBinder(const sp<IBinder>& val)
{
    return flatten_binder(ProcessState::self(), val, this);
}
```

flatten_binder will generate a Binder command. Because BBinder is a local binder object, so the code branch to the lines marked red.

```
status_t flatten_binder(const sp<ProcessState>& proc,
    const sp<IBinder>& binder, Parcel* out)
{
    flat_binder_object obj;

    obj.flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
    if (binder != NULL) {
        IBinder *local = binder->localBinder();
        if (!local) {
            BpBinder *proxy = binder->remoteBinder();
            if (proxy == NULL) {
                LOGE("null proxy");
            }
            const int32_t handle = proxy ? proxy->handle() : 0;
            obj.type = BINDER_TYPE_HANDLE;
            obj.handle = handle;
            obj.cookie = NULL;
        } else {
            obj.type = BINDER_TYPE_BINDER;
            obj.binder = local->getWeakRefs();
            obj.cookie = local;
        }
    }
    return finish_flatten_binder(binder, obj, out);
}
```

Pay attention to the red lines, local’s address is put into the packet(It will be used later). After the packet for addService RPC call is made, BpServiceManager::addService will call BpBinder’s transact.

```
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    // Once a binder has died, it will never come back to life.
    if (mAlive) {
        status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

BpBinder calls IPCThreadState::transact to start a transaction to the binder object corresponding to mHandle. For this scenario, mHandle is 0.

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err = data.errorCheck();

    flags |= TF_ACCEPT_FDS;

    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }
    return err;
}
```

IPCThreadState::transact will first call writeTransactionData to compose a transaction structure for binder kernel driver. Pay attention to the following lines, it’s very important for binder kernel driver to identify the transaction target.

```
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
    int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
{
    binder_transaction_data tr;

    tr.target.handle = handle;
    tr.code = code;
    tr.flags = binderFlags;

    const status_t err = data.errorCheck();
    if (err == NO_ERROR) {
        tr.data_size = data.ipcDataSize();
        tr.data.ptr.buffer = data.ipcData();
        tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
        tr.data.ptr.offsets = data.ipcObjects();
    } else if (statusBuffer) {
        tr.flags |= TF_STATUS_CODE;
        *statusBuffer = err;
        tr.data_size = sizeof(status_t);
        tr.data.ptr.buffer = statusBuffer;
        tr.offsets_size = 0;
        tr.data.ptr.offsets = NULL;
    } else {
        return (mLastError = err);
    }

    mOut.writeInt32(cmd);
    mOut.write(&tr, sizeof(tr));

    return NO_ERROR;
}
```

Then waitForResponse will call talkWithDriver to invoke BINDER_WRITE_READ ioctl.

```
#if defined(HAVE_ANDROID_OS)
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
        else
            err = -errno;
#else
```

So until now the transaction data has been delivered to binder kernel driver.

Summary:
Proxy object will generate the needed packet for a RPC call and then invoke BINDER_WRITE_READ to write the packet to binder kernel driver. The packet is a formatted packet. For RPC call, it uses BC_TRANSACTION packet type.
Assume you want to implement your own service IFunnyTest, you must do the following:

- In the process your service running, call IServiceManager::addService to register the service to servier_manger.

## Transaction in Binder Kernel Driver

When any process opens the “/dev/binder” driver, a corresponding structure binder_proc will be assigned at binder_open.

```
static int binder_open(struct inode *nodp, struct file *filp)
{
        struct binder_proc *proc;

        proc = kzalloc(sizeof(*proc), GFP_KERNEL);
        if (proc == NULL)
                return -ENOMEM;
        get_task_struct(current);
        proc->tsk = current;
        INIT_LIST_HEAD(&proc->todo);
        init_waitqueue_head(&proc->wait);
        proc->default_priority = task_nice(current);
        mutex_lock(&binder_lock);
        binder_stats.obj_created[BINDER_STAT_PROC]++;
        hlist_add_head(&proc->proc_node, &binder_procs);
        proc->pid = current->group_leader->pid;
        INIT_LIST_HEAD(&proc->delivered_death);
        filp->private_data = proc;
        mutex_unlock(&binder_lock);

        if (binder_proc_dir_entry_proc) {
                char strbuf[11];
                snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
                create_proc_read_entry(strbuf, S_IRUGO, binder_proc_dir_entry_proc,
binder_read_proc_proc, proc);
        }

        return 0;
}
```

So that when any ioctl reached, the driver can know the process info. The transaction data is transferred through BINDER_WRITE_READ ioctl.

```
    case BINDER_WRITE_READ: {
        struct binder_write_read bwr;
        if (size != sizeof(struct binder_write_read)) {
            ret = -EINVAL;
            goto err;
        }
        if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
            ret = -EFAULT;
            goto err;
        }
        if (bwr.write_size > 0) {
            ret = binder_thread_write(proc, thread, (void __user *)bwr.write_buffer,
bwr.write_size, &bwr.write_consumed);
            if (ret < 0) {
                bwr.read_consumed = 0;
                if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                    ret = -EFAULT;
                goto err;
            }
        }
        if (bwr.read_size > 0) {
            ret = binder_thread_read(proc, thread, (void __user *)bwr.read_buffer,
bwr.read_size, &bwr.read_consumed, filp->f_flags & O_NONBLOCK);
            if (!list_empty(&proc->todo))
                wake_up_interruptible(&proc->wait);
            if (ret < 0) {
                if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
                    ret = -EFAULT;
                goto err;
            }
        }
        if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
            ret = -EFAULT;
            goto err;
        }
        break;
    }
```

The driver first handles write, then read. Let’s take a look at the binder_thread_write first. The core in binder_thread_write is a loop to parse command from write buffer and execute the corresponding command.

```
    uint32_t cmd;
    void __user *ptr = buffer + *consumed;
    void __user *end = buffer + size;
    while (ptr < end && thread->return_error == BR_OK) {
        if (get_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
        if (_IOC_NR(cmd) < ARRAY_SIZE(binder_stats.bc)) {
            binder_stats.bc[_IOC_NR(cmd)]++;
            proc->stats.bc[_IOC_NR(cmd)]++;
            thread->stats.bc[_IOC_NR(cmd)]++;
        }
        switch (cmd) {
        case ***:

        default:
            printk(KERN_ERR "binder: %d:%d unknown command %d\n", proc->pid,
thread->pid, cmd);
            return -EINVAL;
        }
        *consumed = ptr - buffer;
    }
```

We just take a look at 2 commands related to the scenario. One is BC_INCREFS.

```
        case BC_INCREFS:
        case BC_ACQUIRE:
        case BC_RELEASE:
        case BC_DECREFS: {
            uint32_t target;
            struct binder_ref *ref;
            const char *debug_string;

            if (get_user(target, (uint32_t __user *)ptr))
                return -EFAULT;
            ptr += sizeof(uint32_t);
            if (target == 0 && binder_context_mgr_node &&
                (cmd == BC_INCREFS || cmd == BC_ACQUIRE)) {
                ref = binder_get_ref_for_node(proc,
                                 binder_context_mgr_node);
            } else
                ref = binder_get_ref(proc, target);
            if (ref == NULL) {
                binder_user_error("binder: %d:%d refcou"
                    "nt change on invalid ref %d\n",
                    proc->pid, thread->pid, target);
                break;
            }
```

```
    switch (cmd) {
    case BC_INCREFS:
        debug_string = "IncRefs";
        binder_inc_ref(ref, 0, NULL);
        break;
    }
    break;
```

Remember that, we mentioned it before, in this scenario, the target will be 0. binder_context_mgr_node is created to represent handle 0 when system_manager calls BINDER_SET_CONTEXT_MGR ioctl. So here just increase a weak reference to binder_context_mgr_node node.

```
binder_context_mgr_node = binder_new_node(proc, NULL);
```

The other is BC_TRANSACTION.

```
case BC_TRANSACTION:
case BC_REPLY: {
    struct binder_transaction_data tr;

    if (copy_from_user(&tr, ptr, sizeof(tr)))
        return -EFAULT;
    ptr += sizeof(tr);
    binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
    break;
}
```

binder_transaction will create a new binder node if the packet contains a BINDER_TYPE_BINDER flattened object.

```
fp = (struct flat_binder_object *)(t->buffer->data + *offp);
switch (fp->type) {
case BINDER_TYPE_BINDER:
case BINDER_TYPE_WEAK_BINDER: {
    struct binder_ref *ref;
    struct binder_node *node = binder_get_node(proc, fp->binder);
    if (node == NULL) {
        node = binder_new_node(proc, fp->binder);
        if (node == NULL) {
            return_error = BR_FAILED_REPLY;
            goto err_binder_new_node_failed;
        }
        node->cookie = fp->cookie;
    }
    ref = binder_get_ref_for_node(target_proc, node);
    if (ref == NULL) {
        return_error = BR_FAILED_REPLY;
        goto err_binder_get_ref_for_node_failed;
```

```
            }
            if (fp->type == BINDER_TYPE_BINDER)
                fp->type = BINDER_TYPE_HANDLE;
            else
                fp->type = BINDER_TYPE_WEAK_HANDLE;
            fp->handle = ref->desc;
            binder_inc_ref(ref, fp->type == BINDER_TYPE_HANDLE, &thread->todo);
            if (binder_debug_mask & BINDER_DEBUG_TRANSACTION)
                printk(KERN_INFO "                node %d u%p -> ref %d desc %d\n",
                        node->debug_id, node->ptr, ref->debug_id, ref->desc);
        } break;
```

binder_transaction will know that the target is handle 0, so it runs the following branch to get the target_node, target_proc and target_thread.

```
        } else {
            target_node = binder_context_mgr_node;
            if (target_node == NULL) {
                return_error = BR_DEAD_REPLY;
                goto err_no_context_mgr_node;
            }
        }
        e->to_node = target_node->debug_id;
        target_proc = target_node->proc;
        if (!(tr->flags & TF_ONE_WAY) && thread->transaction_stack) {
            struct binder_transaction *tmp;
            tmp = thread->transaction_stack;
            while (tmp) {
                if (tmp->from && tmp->from->proc == target_proc)
                    target_thread = tmp->from;
                tmp = tmp->from_parent;
            }
        }
```

L1345

Finally binder_transaction will put this request into list and wake up the waiting thread in binder_thread_read.

```
    t->work.type = BINDER_WORK_TRANSACTION;
    list_add_tail(&t->work.entry, target_list);
    tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
    list_add_tail(&tcomplete->entry, &thread->todo);
    if (target_wait)
        wake_up_interruptible(target_wait);
```

在binder_transaction末尾，

Now let’s take a look at binder_thread_read. When service_manager runs, it will wait at here until a request is delivered to it.

```
            ret = wait_event_interruptible_exclusive(proc->wait, binder_has_proc_work(proc, thread));
```

Because the previous write from media_server process has wakened it up, so it gets executed. The following code is to copy data from write buffer of media_server to read buffer of ~~system_manager~~

```
        tr.data_size = t->buffer->data_size;
        tr.offsets_size = t->buffer->offsets_size;
        tr.data.ptr.buffer = (void *)((void *)t->buffer->data + proc->user_buffer_offset);
        tr.data.ptr.offsets = tr.data.ptr.buffer + ALIGN(t->buffer->data_size, sizeof(void *));

        if (put_user(cmd, (uint32_t __user *)ptr))
            return -EFAULT;
        ptr += sizeof(uint32_t);
        if (copy_to_user(ptr, &tr, sizeof(tr)))
            return -EFAULT;
        ptr += sizeof(tr);
```

Summary:

This section has shown the data flow from RPC call client side to RPC server side.

## Service Manager Handle Add Service

Until now, the service_manager has gotten a packet of BR_TRANSACTION from media_server, so it will call binder_parser to handle the packet.

```
        case BR_TRANSACTION: {
            if (func) {
                unsigned rdata[256/4];
                struct binder_io msg;
                struct binder_io reply;
                int res;

                bio_init(&reply, rdata, sizeof(rdata), 4);
                bio_init_from_txn(&msg, txn);
                res = func(bs, txn, &msg, &reply);
                binder_send_reply(bs, &reply, txn->data, res);
            }
            ptr += sizeof(*txn) / sizeof(uint32_t);
            break;
        }
```

binder_parser will call svcmgr_handler to parse BR_TRANSACTION packet, which is the reverse process of BpServerManager. Here structure binder_txn actually is the same with structure binder_transaction_data. In our scenario, the transaction code is SVC_MGR_ADD_SERVICE.

```
int svcmgr_handler(struct binder_state *bs,
                   struct binder_txn *txn,
                   struct binder_io *msg,
                                   struct binder_io *reply)
{
        struct svcinfo *si;
        uint16_t *s;
        unsigned len;
        void *ptr;

        if (txn->target != svcmgr_handle)
                return -1;

        s = bio_get_string16(msg, &len);

        if ((len != (sizeof(svcmgr_id) / 2)) ||
                memcmp(svcmgr_id, s, sizeof(svcmgr_id))) {
                fprintf(stderr,"invalid id %s\n", str8(s));
                return -1;
        }

        switch(txn->code) {
        case SVC_MGR_ADD_SERVICE:
                s = bio_get_string16(msg, &len);
                ptr = bio_get_ref(msg);
                if (do_add_service(bs, s, len, ptr, txn->sender_euid))
                        return -1;
                break;
```

So that service_manager knows that a service named s will run and it gets the object information through bio_get_ref.

```
void *bio_get_ref(struct binder_io *bio)
{
        struct binder_object *obj;

        obj = _bio_get_obj(bio);
        if (!obj)
                return 0;

        if (obj->type == BINDER_TYPE_HANDLE)
                return obj->pointer;

        return 0;
}
```

bio_get_ref is the reverse work of flatten_binder. do_add_service will finally call BC_ACQUIRE to get a strong reference to the object represented by ptr.

Summary:

This section shows how the service is added to service manager.

Assume you want to implement your own service IFunnyTest, you must do the following:

- Add your service name into allowed service list in service_manager.

## Get IAudioFlinger

The only way to get a service interface is through IServiceManager::getService. For example, here is the method for AudioSystem to get an IAudioFlinger.

```
// establish binder interface to AudioFlinger service
const sp<IAudioFlinger>& AudioSystem::get_audio_flinger()
{
    Mutex::Autolock _l(gLock);
    if (gAudioFlinger.get() == 0) {
        sp<IServiceManager> sm = defaultServiceManager();
        sp<IBinder> binder;
        do {
            binder = sm->getService(String16("media.audio_flinger"));
            if (binder != 0)
                break;
            LOGW("AudioFlinger not published, waiting...");
            usleep(500000); // 0.5 s
        } while(true);
        gAudioFlinger = interface_cast<IAudioFlinger>(binder);
    }
    LOGE_IF(gAudioFlinger==0, "no AudioFlinger!?");
    return gAudioFlinger;
}
```

参照Generate AudioFlinger Service

IServiceManager::getService is called to BpServiceManager::getService.

```
    virtual sp<IBinder> getService(const String16& name) const
    {
        unsigned n;
        for (n = 0; n < 5; n++){
            sp<IBinder> svc = checkService(name);
            if (svc != NULL) return svc;
            LOGI("Waiting for sevice %s...\n", String8(name).string());
            sleep(1);
        }
        return NULL;
    }
    virtual sp<IBinder> checkService( const String16& name) const
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        data.writeString16(name);
        remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);
        return reply.readStrongBinder();
    }
```

Just like analyzed before, the call will finally through Binder kernel driver to be handled in service_manager process.

```
    switch(txn->code) {
    case SVC_MGR_GET_SERVICE:
    case SVC_MGR_CHECK_SERVICE:
        s = bio_get_string16(msg, &len);
        ptr = do_find_service(bs, s, len);
        if (!ptr)
            break;
        bio_put_ref(reply, ptr);
        return 0;
```

Then service_manager will reply with the previous handle set by media_server (Which is the AudioFlinger's instance's address.). Then BpServiceManager::checkService will be returned back from remote()->transact call. Then just like analyzed for IServiceManager, it will new another BpBinder instance corresponding to the handle returned back from service_manager. interface_cast<IAudioFlinger>(binder) will finally return an BpAudioFlinger instance.

Summary:

Just like get IServieManager, but this time it needs to get a handle from service_manager. While for IServiceManager it always use handle 0.

# RPC Call IAudioFlinger::SetMode

If we call IAudioFlinger::SetMode in AAA process, actually call BpAudioFlinger::setMode.

```
    virtual status_t setMode(int mode)
    {
        Parcel data, reply;
        data.writeInterfaceToken(IAudioFlinger::getInterfaceDescriptor());
        data.writeInt32(mode);
        remote()->transact(SET_MODE, data, &reply);
        return reply.readInt32();
    }
```

Like analysis of IServiceManager::addService, this function will generate a packet and write it to binder kernel driver, then wait to read reply. The only difference is this time the target handle points to some address in media_server process.

## Handle IAudioFlinger::SetMode

The Binder kernel driver will finally wake up the read thread of media_server process which runs at IPCThreadState::joinThreadPool. Now let’s re-look the code.

```
void IPCThreadState::joinThreadPool(bool isMain)
{
    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);

    status_t result;
    do {
        int32_t cmd;
        result = talkWithDriver();
        if (result >= NO_ERROR) {
            size_t IN = mIn.dataAvail();
            if (IN < sizeof(int32_t)) continue;
            cmd = mIn.readInt32();
            result = executeCommand(cmd);
        }

        // Let this thread exit the thread pool if it is no longer
        // needed and it is not the main process thread.
        if(result == TIMED_OUT && !isMain) {
            break;
        }
    } while (result != -ECONNREFUSED && result != -EBADF);

    mOut.writeInt32(BC_EXIT_LOOPER);
    talkWithDriver(false);
}
```

At this time, talkWithDriver will return with the packet generated by BpServiceManager::setMode, then executeCommand will process the commands. In this scenario, the command is BR_TRANSACTION.

```
    case BR_TRANSACTION:
        {
            binder_transaction_data tr;

            Parcel reply;
            if (tr.target.ptr) {
                sp<BBinder> b((BBinder*)tr.cookie);
                const status_t error = b->transact(tr.code, buffer, &reply, 0);
                if (error < NO_ERROR) reply.setError(error);

            } else {
                const status_t error = the_context_object->transact(tr.code, buffer, &reply, 0);
                if (error < NO_ERROR) reply.setError(error);
            }

            if ((tr.flags & TF_ONE_WAY) == 0) {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
                sendReply(reply, 0);
            } else {
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
            }
        }
        break;
```

The most important two lines are marked with red. Here it gets an address from binder kernel driver and cast it to a BBinder pointer(This address in put into binder kernel driver when called IServiceManager::addService.). Remember that, AudioFlinger is derived from BBinder. This pointer is actually the same pointer of our AudioFlinger instance. So the following transact call will finally call our BnAudioFlinger's onTransact virtual function.

```
status_t BnAudioFlinger::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
        case SET_MODE: {
            CHECK_INTERFACE(IAudioFlinger, data, reply);
            int mode = data.readInt32();
            reply->writeInt32( setMode(mode) );
            return NO_ERROR;
        } break;
}
```

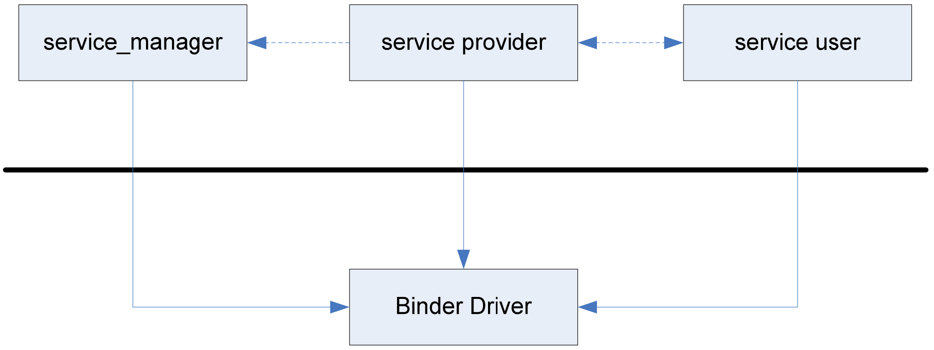
Then the reply will be written through sendReply.

```
status_t IPCThreadState::sendReply(const Parcel& reply, uint32_t flags)
{
    status_t err;
    status_t statusBuffer;
    err = writeTransactionData(BC_REPLY, flags, -1, 0, reply, &statusBuffer);
    if (err < NO_ERROR) return err;

    return waitForResponse(NULL, NULL);
}
```

It will finally write to binder kernel driver. Then kernel driver will wake up the read thread of AAA.

# Summary



The overall architecture of Android IPC system is shown in the picture. There are four major blocks:

- Binder Driver

  It's the core of IPC system. It passes data between service provider and service user.
- service provider

  It provides some kind of service. It will parser the received RPC call data from binder driver and do the real action.
- service_manager

  It's a special service provider. It provides service manager service for other service provider.
- service user

  It remote calls service provider. It will generate an RPC call data and send it to binder driver.

For the example scenario, here lists the major control flow.

1. service_manager runs first, it will register a special node 0 in binder driver.
2. media_server gets an IServiceManager proxy object for the special node 0.
3. media_server RPC call IServiceManager::addService to add the IAudioFlinger service. This call is routed to node 0. It will send data to binder driver.
4. binder driver notify that data is for node 0, and the data contains command to binder a object. So it will generate another node (assume A) for the IAudioFlinger service and route the data to service_manager.
5. service_manager reads data from binder driver, then process IServiceManager::addService RPC call.
6. another process P gets an IServiceManager proxy object for the special node 0.
7. P RPC call IServiceManager::getService to get IAudioFlinger service. This call is routed to node 0. It will send data to binder driver.
8. binder driver notify that data is for node 0, So it will route the data to service_manager.
9. service_manager reads data from binder driver, then process IServiceManager::getService RPC call and return back the node A to represent the IAudioFlinger service.
10. P RPC call IAudioFlinger::setMode. Now this call will be routed to node A.
11. binder driver notify that data is for node A, so it will route the data to media_server.
12. media_server reads data from binder driver, handles IAudioFlinger::setMode RPC call and send reply data to binder driver.
13. binder driver route the reply to P.
14. P reads data from binder driver and finally gets the reply.