# Introduction

**Process in Android have separate address spaces**, so one cannot directly access another process memory. It’s called process isolation – it’s good for the security and stability of the OS.

However suppose a process want to offer some useful services to other processes, in that case it **needs to provide some mechanism that allows other processes to discover and interact with those services.**

We know that Android is built on top of Linux kernel. But it removes lots of IPC mechanisms available in Linux. Instead, Android has its own IPC mechanisms specific to the OS, such as:

* Binder
* Intents
* Content Providers
* Messenger
* …

They are designed to **address the specific architecture and characteristics of the Android platform**, including inter-component communication, data sharing, and security considerations.

# Binder

## What Is Binder?

Binder is a high-level mechanism for transferring data between processes. In Android OS, Binder allows Android services and apps to communicate with each other by sending and receiving messages through a centralized message queue.

By being high-level, Binder abstracts the low-level details of IPC.

Although Binder is native to Android, it can be ported to be used in Linux. You can check this [experimental project](https://github.com/hungys/binder-for-linux).

Origin:

Binder is developed by Google based on OpenBinder and target to the Android OS. Starting in Android 8, Android framework and HALs now communicate with each other using Binder.

The below diagram shows IPC location in the Android system:

|  |  |
| --- | --- |
|  |  |

## Why Binder?

Traditional Linux IPCs (particularly, System V) have drawbacks that make it unsuitable for being used in Android:

* Android uses LowMemoryKiller – a component responsible for killing processes under low-memory conditions. The problem is that some **System V IPCs (including shared memory, message queue, etc.) are prone to Kernel resource leakage**; when a process is killed forcefully, it doesn’t release shared IPC resources upon termination.

🡺 Actually, Android (particularly, its libC – *Bionic*) **does NOT support some System V IPCs**, including semaphores, shared memory, message queues, etc.

Binder was born to fix above issues and offer various advantages.

You might not know!

In addition to using Binder, we have **other ways to address the release leak issue of System V IPC**, such as:

* **Use POSIX IPCs**: Because of using reference counting technique, they automatically release resources when the last process using them terminates.
* **Use signal handling**: Processes can register signal handlers to catch termination signals (e.g., SIGTERM or SIGINT) and perform cleanup operations before exiting.
* **Use system administrator intervention:** It can periodically monitor and clean up unused System V IPC resources. Tools such as *ipcs* and *ipcrm* in Linux can be used to list and remove lingering IPC resources.

## Pros and Cons

**Pros:**

* Support complex data structure: file descriptor, shared memory, weak/strong references.
* Binder = light-weight RPC (but in the same machine)
* "Thread migration" - like programming model:
  + Automatic management of thread-pools
* Methods on remote objects can be invoked as if they were local - the thread appears to "jump" to the other process
* Synchronous and asynchronous (oneway) invocation model
* Identifying senders to receivers (via UID/PID) - important for security reasons.
* Unique object-mapping across process boundaries: A reference to a remote object can be passed to another process and can be used as an identifying token.
* Ability to send file descriptors across process boundaries.
* Simple Android Interface Definition Language (AIDL).
* Built-in support for marshalling many common data-types.
* Simplified transaction invocation model via auto-generated proxies and stubs (Java-only).
* Recursion across processes - i.e. behaves the same as recursion semantics when calling methods on local objects.
* Local execution mode (no IPC/data marshalling) if the client and the service happen to be in the same process.

**Cons:**

* Binders are **not good for transferring large data streams** (like video/audio) because every object has to be serialized to a parcel and de-serialized from the parcel.
* Not defined by POSIX or any other standard.
* Binder is limited to 1 MB per transaction.
* Binder thread pool is limited to 15 per process.

## How Binder Works?

### Proxy-Stub Class Implementation

If we consider from user space, binder works based on a *Proxy-Stub* class implementation.

Following is Binder’s **workflow**:

|  |  |
| --- | --- |
|  | *Proxy* class helps client process to request the server process, while *Stub* class helps receive and process these requests.  A request data will be **serialized** to a parcel. Then, the parcel is passed to Binder driver at the kernel by transact messages which uses IOCTL calls underneath.  The parcel will be **deserialized** at the server process to retrieve the data which then is sent to the Binder stub. |

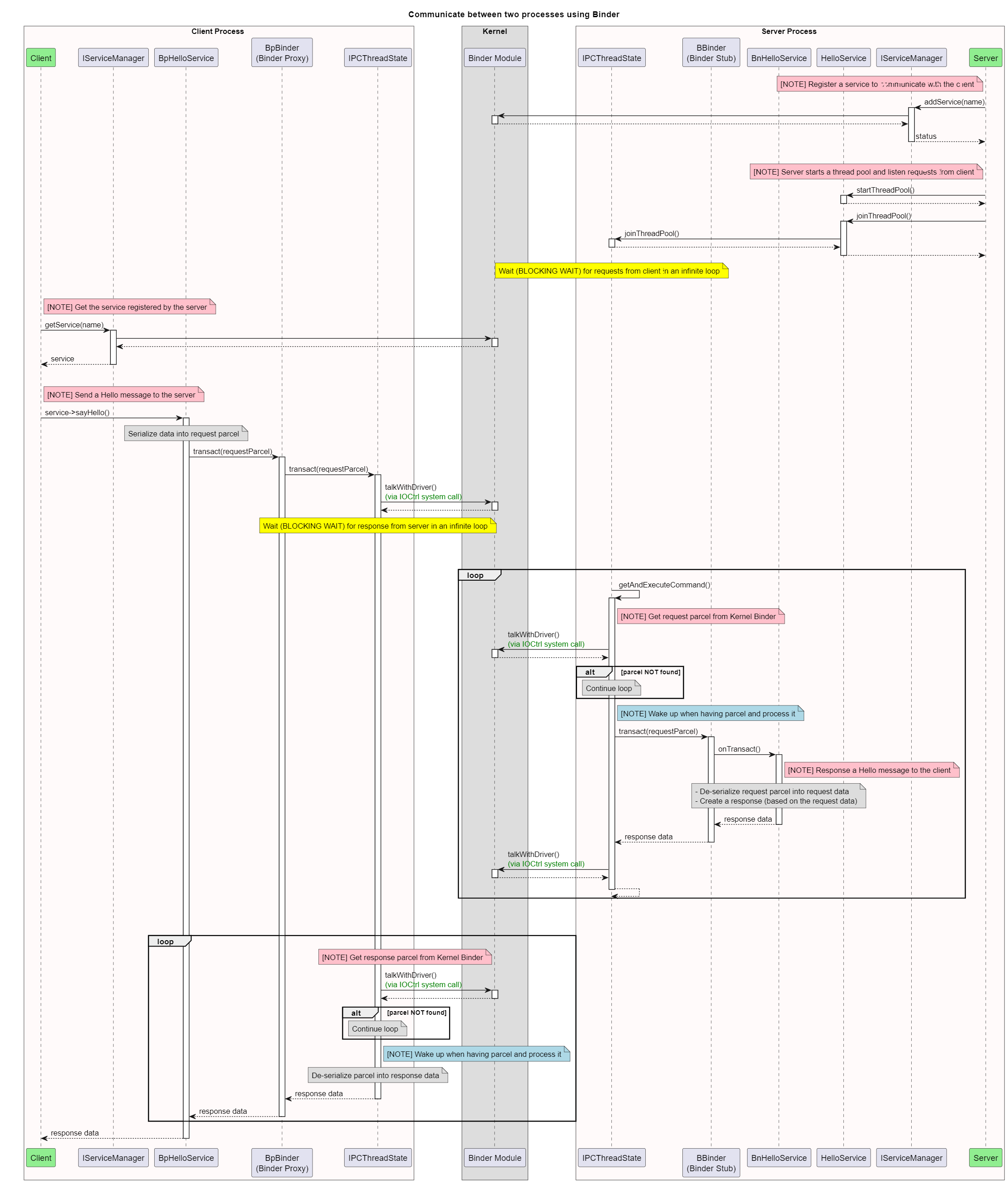
Terms:

* **Serialize (**or **marshal)**: The action of placing the parcel into a continuous buffer memory.
* **Deserialize (**or **unmarshal)**: The action of retrieve the parcel from a continuous buffer memory.
* **Transact**: The action sending a Binder message. For a Binder message, we call it a "transaction".
* **Parcel**: The data payload in a transaction.
* **IOCTL** (Input/Output Control): A system call allowing user-space programs to communicate with device drivers and other kernel subsystems.
* **IPC Thread**: A thread for handling transactions. It does lots of things: writing transaction data, waiting for response from server, sending a response from server to client, interacting with the kernel, etc.

Why need a proxy and a stub?

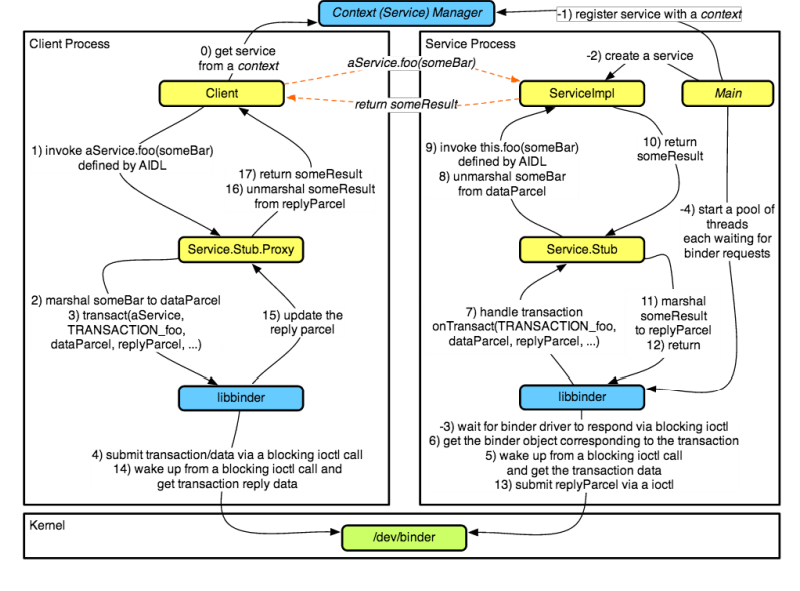
* They are **simply wrappers that hide the implement details** of the interface.
* For the client side, we name that wrapper a Proxy. For the server side, we name it a Stub (Stub is actually Proxy).

Following is Binder’s **sequence diagram**. We’ll discover how the client sends a Hello request to the server and how the server sends a Hello response to the client:



For the **sequence diagram’s script** and **source code example**, check here: *Tutorials\Linux\Code\IPCs\Binder*

**Detailed workflow reference:**



### Reference Counting

Binder uses **reference counting** technique for object references. This makes it suitable for "hostile" environments where LowMemoryKiller roams.

**Object references are remembered by the Binder kernel driver**. Every time a reference is shared with another process, its reference count is incremented. The reference count is decremented either explicitly or automatically when the process dies. When a reference is no longer needed, its owner is notified that it can be released, so Binder removes its mapping.

Even when the client is killed or crash suddenly, the shared resources will still be cleared.

### Scatter-Gather Mechanism

The scatter-gather mechanism can be **a read procedure from multiple data streams and write to a single data stream**, or **a read procedure from a single stream and write into multiple streams**.

If we consider it with large chunk of data, instead of using reference to large data, we can use multiple references of its small streams to carry the data. Generally scatter gather mechanism is described with non-contiguous memory locations. The mechanism emerges as an efficient way to carry large-sized data streams.

In previous releases of Android, every piece of data in a binder call was copied three times:

* Once to serialize it into a Parcel in the calling process
* Once in the kernel driver to copy the Parcel to the target process
* Once to unserialize the Parcel in the target process

Android 8 uses scatter-gather optimization to reduce the number of copies from 3 to 1. Instead of serializing data in a Parcel first, data remains in its original structure and memory layout and the driver immediately copies it to the target process. After the data is in the target process, the structure and memory layout is the same and the data can be read without requiring another copy.

### Death Recipient (or Death-Notification Mechanism)

It’s a mechanism used to **monitor the liveness of a remote process and receive notifications when it dies**.

In Android, any process (e.g., process A) can receive a notification when another process (e.g., process B) dies by taking the following steps:

1. Process A creates a DeathRecipient callback object containing the code to be executed when the death notification arrives.
2. Process A obtains a reference to a Binder object that lives in process B and calls its linkToDeath(), passing the DeathRecipient callback object as the first argument.
3. Process A waits for process B (that hosts the Binder object) to die. When the Binder kernel driver detects that process B is gone, it’ll notify the registered DeathRecipient callback object by calling its binderDied().

Notes:

* In Binder framework, the interface is the /dev/binder device. It’s implemented by the Binder kernel driver which is the central object of the framework, and all IPC calls go through it.
* Bn\*\*\* (Binder native objects): They’re objects implemented in native code and **used by the server to expose their services to the client**. They’re created by inheriting from the BBinder class and implementing the onTransact().

The onTransact() method is called by the Binder when a client invokes a method on the Bn object.

* Bp\*\*\* (Binder proxy objects): They’re objects **used by the client to access services provided by the server**. They’re generated by the system at runtime based on an interface definition provided by the server. They provide an interface that can be used by client processes to invoke methods on the server.
* Bp will work as a "proxy". Bn will work as a "Stub".

OTHER INFO:

Binder object and interface:

Binder Tokens

An interesting property of Binder object is that each instance maintains a unique identity across all processes in the system, no matter how many process boundaries it crosses or where it goes. This facility is provided by the Binder kernel driver as explained above.

Each Binder’s object reference is assigned either:

* A virtual memory address pointing to a Binder object in the same process,

or

* A unique 32-bit handle (as assigned by the Binder kernel driver) pointing to the Binder’s virtual memory address in a different process.

The Binder kernel driver maintains a mapping of local addresses to remote Binder handles (and vice versa) for each Binder it sees, and assigns each Binder’s object reference its appropriate value to ensure that equality will behave as expected even in remote processes. The Binder’s unique object identity rules allow them to be used for a special purpose: as shared, security access tokens.

How to implement Native and Proxy Binder object

1. IBinder

In native user space the IBinder is the base class of all kinds of Binder objects, both proxy objects and native objects. All native Binder objects inherit from BBinder and all proxy Binder objects inherit from BpBinder

The most important method of this interface is transact( uint32\_t code,

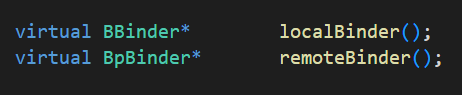
const Parcel& data,

Parcel\* reply,

uint32\_t flags = 0) = 0;

* code is the transaction code of this transaction
* data is the input data and reply is the output for this transaction
* The most important bit mask for flags argument is FLAG\_ONEWAY by setting which you make this transaction asynchronous

Use localBinder and remoteBinder to "cast" this object to a native object and proxy object respectively. For BpBinder, localBinder returns null and remoteBinder returns itself; for BBinder, localBinder returns itself and remoteBinder returns null



2. BBinder

BBinder is the base class of all native Binder objects. It is the class that implements the actual transaction business logic

The BBinder adds a new method onTransact to the IBinder interface. Subclasses of BBinder should override onTransact to implement the actual transaction logic

The transact method handles certain types of transaction codes that are common to all BBinder instances. Otherwise onTransact is called to handle service specific transaction codes that subclass of BBinder should implement

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;

}

3. BpBinder

BpBinder is the base class of all proxy Binder objects, basically it is a reference to a native Binder objects running in another process. Proxy objects don't implement the actual transaction business logic, they just delegate the call to the underlying BBinder in a remote process with the help of Binder driver

The BpBinder adds a bunch of death notification related classes and methods. BpBinder needs to have the capability of listening to the death of a remote BBinder

A new field mHandle is added in BpBinder. This field is used for distinguishing the same proxy from from different process.

BpBinder invokes a transact method on IPCThreadState to initiate a Binder transaction with Binder driver. The driver will eventually causes the transact method on the target BBinder to be called and gets the result back in reply parameter

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;

}

The linkToDeath also uses IPCThreadState to register a listener for the remote BBinder object. BpBinder will callback into every DeathRecipient registered.

How to implement Native and Proxy Binder Interface

BBinder is a dummy native Binder object that can be discovered by other processes. A BpBinder is a dummy proxy Binder object that is capable of referencing a BBinder in a remote process. They don't implement any custom service interfaces. In order to implement a service that's discoverable by another process, we need to use BnInterface class. In order to implement a service proxy that's capable of referencing a remote BnInterface and delegating service calls to the remote BnInterface we need to use BpInterface

1. IInterface

IInterface is the base class any custom service provider should extend from.

We can use aidl-cpp tool to generate an interface from aidl file

2. BnInterface

The BnInterface class inherits from BBinder so it gains the capability of being referenced remotely. It inherits from INTERFACE so it inherits the service methods listed in INTERFACE: class BnInterface : public INTERFACE, public BBinder

The INTERFACE generic type must be a subclass of IInterface.

For example BnNGTPManagerService: class BnNGTPManagerService : public BnInterface<INGTPManagerService>, the INGTPManagerService is inherited from IInterface: class INGTPManagerService : public IInterface

3. BpInterface

The BpInterface inherits from a new class BpRefBase, which have a mRemote to point to the corresponding remote BBinder

It also inherit from INTERFACE. But BpInterface will just delegate all calls to mRemote who will then use Binder driver to pass the calls to the remote BnInterface

The BpInterface can't inherit from BpBinder because the connection between BBinder and BpBinder is controlled by Binder driver, and it know nothing about the Binder service information. After a BBinder passes through process boundary, all that a remote process receives is the integer handle value. All information this handle value carries is that it references a remote BBinder. No Binder service information is in it so Binder framework cannot initialize a BpInterface from the handle.‌

Send/ Receive Implementation

- Each client of the driver has 1 or more threads.

- A thread on the server waits on a loop on an ioctl waiting for a service request

- The driver puts the thread to sleep using wait\_event\_interruptible

- When an app calls ioctl on its end targeting a service, the driver wakes up a thread of that service

- ioctl on service end, comes out of the wait, services the request

- Now, if it's a Sync request, app makes another ioctl call waiting for reply

- The services sends a reply parcel back by calling ioctl, waking up the app, and goes back to sleep with another ioctl call (typically in a loop)

- If the request is Async, service calls ioctl sometime later. But this time, one of the threads waiting with ioctl will pick it up

