Deep Dive into Android IPC/Binder Framework at Android Builders Summit 2013

Aleksandar (Saša) Gargenta, Marakana Inc.
Why are you here?

- You want to better understand how Android works
  - Intents, ContentProviders, Messenger
  - Access to system services
  - Life-cycle call-backs
  - Security

- You want to modularize your own business logic
  across application boundaries via a highly efficient
  and low-latency IPC framework

- You want to add new system services and would like to learn how to best expose them to your developers

- You just care about IPC and Binder seems unique and interesting

- You don’t have anything better to do?
Objectives

- Binder Overview
- IPC
- Advantages of Binder
- Binder vs Intent/ContentProvider/Messenger-based IPC
- Binder Terminology
- Binder Communication and Discovery
- AIDL
- Binder Object Reference Mapping
- Binder by Example
- Async Binder
- Memory Sharing
- Binder Limitations
- Security

Slides and screencast from this class will be posted to: http://mrkn.co/bgnhg
Who am I?

Aleksandar Gargenta

- Developer and instructor of Android Internals and Security training at Marakana
- Founder and co-organizer of San Francisco Android User Group (sfandroid.org)
- Founder and co-organizer of San Francisco Java User Group (sfjava.org)
- Co-founder and co-organizer of San Francisco HTML5 User Group (sfhtml5.org)
- Speaker at AnDevCon, AndroidOpen, Android Builders Summit, etc.
- Server-side Java and Linux, since 1997
- Android/embedded Java and Linux, since 2009
- Worked on SMS, WAP Push, MMS, OTA provisioning in previous life
- Follow
  - @agargenta
  - +Aleksandar Gargenta
  - http://marakana.com/s/author/1/aleksandar_gargenta
What is Binder?

- An IPC/component system for developing object-oriented OS services
  - Not yet another *object-oriented kernel*
  - Instead an *object-oriented operating system environment* that works on traditional kernels, like Linux!

- Essential to Android!
- Comes from OpenBinder
  - Started at Be, Inc. as a key part of the "next generation BeOS" (~ 2001)
  - Acquired by PalmSource
  - First implementation used in Palm Cobalt (micro-kernel based OS)
  - Palm switched to Linux, so Binder ported to Linux, open-sourced (~ 2005)
  - Google hired Dianne Hackborn, a key OpenBinder engineer, to join the Android team
  - Used as-is for the initial bring-up of Android, but then completely rewritten (~ 2008)
  - OpenBinder no longer maintained - long live Binder!

- Focused on scalability, stability, flexibility, low-latency/overhead, easy programming model
IPC

- Inter-process communication (IPC) is a framework for the exchange of signals and data across multiple processes
- Used for message passing, synchronization, shared memory, and remote procedure calls (RPC)
- Enables information sharing, computational speedup, modularity, convenience, privilege separation, data isolation, stability
  - Each process has its own (sandboxed) address space, typically running under a unique system ID

- Many IPC options
  - Files (including memory mapped)
  - Signals
  - Sockets (UNIX domain, TCP/IP)
  - Pipes (including named pipes)
  - Semaphores
  - Shared memory
  - Message passing (including queues, message bus)
  - Intents, ContentProviders, Messenger
  - Binder!
**Why Binder?**

- Android apps and system services run in separate processes for security, stability, and memory management reasons, but they need to communicate and share data!
  - Security: each process is sandboxed and run under a distinct system identity
  - Stability: if a process misbehaves (e.g. crashes), it does not affect any other processes
  - Memory management: "unneeded" processes are removed to free resources (mainly memory) for new ones
  - In fact, a single Android app can have its components run in separate processes

- IPC to the rescue
  - But we need to avoid overhead of traditional IPC and avoid denial of service issues

- Android’s libc (a.k.a. bionic) does not support System V IPCs,
  - No SysV semaphores, shared memory segments, message queues, etc.
  - System V IPC is prone to kernel resource leakage, when a process "forgets" to release shared IPC resources upon termination
  - Buggy, malicious code, or a well-behaved app that is low-memory SIGKILL’ed

- Binder to the rescue!
  - Its built-in reference-counting of "object" references plus death-notification mechanism make it suitable for "hostile" environments (where lowmemorykiller roams)
  - When a binder service is no longer referenced by any clients, its owner is automatically notified that it can dispose of it

- Many other features:
  - "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 yet 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

- But:
  - No support for RPC (local-only)
  - Client-service message-based communication - not well-suited for streaming
  - Not defined by POSIX or any other standard

- Most apps and core system services depend on Binder
  - Most app component life-cycle call-backs (e.g. onResume(), onDestroy(), etc.) are invoked by ActivityManagerService via binder
  - Turn off binder, and the entire system grinds to a halt (no display, no audio, no input, no sensors, ...)
  - Unix domain sockets used in some cases (e.g. RILD)
IPC with Intents and ContentProviders?

- Android supports a simple form of IPC via intents and content providers
- Intent messaging is a framework for asynchronous communication among Android components
  - Those components may run in the same or across different apps (i.e. processes)
  - Enables both point-to-point as well as publish-subscribe messaging domains
  - The intent itself represents a message containing the description of the operation to be performed as well as data to be passed to the recipient(s)
  - Implicit intents enable loosely-coupled APIs

- ContentResolvers communicate synchronously with ContentProviders (typically running in separate apps) via a fixed (CRUD) API
- All android component can act as a senders, and most as receivers
- All communication happens on the Looper (a.k.a. main) thread (by default)
- But:
  - Not really OOP
  - Asynchronous-only model for intent-based communication
  - Not well-suited for low-latency
  - Since the APIs are loosely-defined, prone to run-time errors
  - All underlying communication is based on Binder!
  - In fact, Intents and ContentProvider are just a higher-level abstraction of Binder
  - Facilitated via system services: ActivityManagerService and PackageManagerService

- For example:
  src/com/marakana/shopping/UpcLookupActivity.java
public class ProductLookupActivity extends Activity {
    private static final int SCAN_REQ = 0;

    public void onClick(View view) {
        Intent intent = new Intent("com.google.zxing.client.android.SCAN");  // 1
        intent.setPackage("com.google.zxing.client.android");               // 2
        intent.putExtra("SCAN_MODE", "PRODUCT_MODE");                    // 3
        super.startActivityForResult(intent, SCAN_REQ);                   // 3
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {  // 4
        if (requestCode == SCAN_REQ && resultCode == RESULT_OK) {             // 5
            String barcode = data.getStringExtra("SCAN_RESULT");            // 6
            String format = data.getStringExtra("SCAN_RESULT_FORMAT");     // 6

            super.startActivity(
                new Intent(Intent.ACTION_VIEW,
                Uri.parse("http://www.upcdatabase.com/item/" + barcode)));
            // 7
        }
        ...
    }
}

src/com/google/zxing/client/android/CaptureActivity.java:

public class CaptureActivity extends Activity {

    private void handleDecodeExternally(Result rawResult, ...) {
        Intent intent = new Intent(getIntent().getAction());                // 8
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET);
        intent.putExtra(Intents.Scan.RESULT, rawResult.toString());         // 9
        intent.putExtra(Intents.Scan.RESULT_FORMAT, 
            rawResult.getBarcodeFormat().toString());                     // 8

        super.setResult(Activity.RESULT_OK, intent);                       // 9
        super.finish();                                                    // 9
    }
}

1. Specify who we want to call
2. Specify the input parameter for our call
3. Initiate the call asynchronously
4. Receive the response via a call-back
5. Verify that this is the response we were expecting
6. Get the response
7. Initiate another IPC request, but don’t expect a result
8. On the service side, put the result into a new intent
9. Send the result back (asynchronously)
Android's **Messenger** represents a reference to a **Handler** that can be sent to a remote process via an **Intent**

A reference to the **Messenger** can be sent via an **Intent** using the previously mentioned IPC mechanism

**Messages** sent by the remote process via the messenger are delivered to the local handler

**Messages** are like **Intents**, in that they can designate the "operation" (**aMessage.what**) and data (**aMessage.getData()**) still asynchronous, but lower latency/overhead

Great for efficient call-backs from the service to the client

Messages are by default handled on the Looper thread

All underlying communication is still *based on Binder*

For example, this is how we could write a client:

```
src/com/marakana/android/download/client/DownloadClientActivity.java:
```
public class DownloadClientActivity extends Activity {
    private static final int CALLBACK_MSG = 0;
    ...
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(
            "com.marakana.android.download.service.SERVICE"); // 1
        ArrayList<String> uris = ...
        intent.putExtra("uris", uris); // 2
        Messenger messenger = new Messenger(new ClientHandler(this)); // 3
        intent.putExtra("callback-messenger", messenger); // 4
        super.startService(intent); // 5
    }

    private static class ClientHandler extends Handler {
        private final WeakReference<DownloadClientActivity> clientRef; // 6

        public ClientHandler(DownloadClientActivity client) {
            this.clientRef = new WeakReference<DownloadClientActivity>(client);
        }

        @Override
        public void handleMessage(Message msg) { // 7
            Bundle data = msg.getData();
            DownloadClientActivity client = clientRef.get();
            if (client != null && msg.what == CALLBACK_MSG && data != null) {
                Uri completedUri = data.getString("completed-uri"); // 8
                // client now knows that completedUri is done
                ...
            }
        }
    }
}

1 Specify who we want to call (back to using Intents!)
2 Specify the input parameter for our call
3 Create a messenger over our handler
4 Pass the messenger also as an input parameter
5 Initiate the call asynchronously
6 Our handler remembers a reference to the client
7 Receive responses via a call-back on the handler
8 Get the response data

- And our service could look as follows:

src/com/marakana/android/download/service/DownloadService.java:
public class MessengerDemoService extends IntentService {
    private static final int CALLBACK_MSG = 0;

    @Override
    protected void onHandleIntent(Intent intent) {
        ArrayList<Uri> uris = intent.getParcelableArrayListExtra("uris");
        Messenger messenger = intent.getParcelableExtra("callback-messenger");
        for (Uri uri : uris) {
            // download the uri
            if (messenger != null) {
                Message message = Message.obtain();
                message.what = CALLBACK_MSG;
                Bundle data = new Bundle(1);
                data.putParcelable("completed-uri", uri);
                message.setData(data);
                try {
                    messenger.send(message);
                } catch (RemoteException e) {
                    ...
                }
            } finally {
                message.recycle();
            }
        }
    }
}

1. Handle the request from our client (which could be local or remote)
2. Get the request data
3. Get the reference to the messenger
4. Use Message as a generic envelope for our data
5. Set our reply
6. Send our reply
Binder Terminology

Binder (Framework)
The overall IPC architecture

Binder Driver
The kernel-level driver that fascinates the communication across process boundaries

Binder Protocol
Low-level protocol (ioctl-based) used to communicate with the Binder driver

IBinder Interface
A well-defined behavior (i.e. methods) that Binder Objects must implement

AIDL
Android Interface Definition Language used to describe business operations on an IBinder Interface

Binder (Object)
A generic implementation of the IBinder interface

Binder Token
An abstract 32-bit integer value that uniquely identifies a Binder object across all processes on the system

Binder Service
An actual implementation of the Binder (Object) that implements the business operations

Binder Client
An object wanting to make use of the behavior offered by a binder service

Binder Transaction
An act of invoking an operation (i.e. a method) on a remote Binder object, which may involve sending/receiving data, over the Binder Protocol

Parcel
"Container for a message (data and object references) that can be sent through an IBinder." A unit of transactional data - one for the outbound request, and another for the inbound reply

Marshalling
A procedure for converting higher level applications data structures (i.e. request/response parameters) into parcels for the purposes of embedding them into Binder transactions
Unmarshalling
A procedure for reconstructing higher-level application data-structures (i.e. request/response parameters) from parcels received through Binder transactions

Proxy
An implementation of the AIDL interface that un/marshals data and maps method calls to transactions submitted via a wrapped IBinder reference to the Binder object

Stub
A partial implementation of the AIDL interface that maps transactions to Binder Service method calls while un/marshalling data

Context Manager (a.k.a. servicemanager)
A special Binder Object with a known handle (registered as handle 0) that is used as a registry/lookup service for other Binder Objects (name → handle mapping)
Binder Communication and Discovery

- As far as the client is concerned, it just wants to use the service:

- While processes cannot directly invoke operations (or read/write data) on other processes, the kernel can, so they make use of the Binder driver:

⚠️ Since the service may get concurrent requests from multiple clients, it needs to protect (synchronize access to) its mutable state.

- Binder driver is exposed via /dev/binder and offers a relatively simple API based on open, release, poll, mmap, flush, and ioctl operations.
- In fact most communication happens via ioctl(binderFd, BINDER_WRITE_READ, &bwd), where bwd is defined as:
struct binder_write_read {
    signed long write_size; /* bytes to write */
    signed long write_consumed; /* bytes consumed by driver */
    unsigned long write_buffer;
    signed long read_size; /* bytes to read */
    signed long read_consumed; /* bytes consumed by driver */
    unsigned long read_buffer;
};

- The `write_buffer` contains a series of commands for the driver to perform
  - Book-keeping commands, e.g. inc/decrement binder object references, request/clear death notification, etc.
  - A command requiring a response, like `BC_TRANSACTION`

- Upon returning, the `read_buffer` will contain commands for the user-space to perform
  - Same book-keeping commands
  - A command requesting processing of the response (i.e. `BC_REPLY`) or a request to perform a nested (recursive) operation

- Clients communicate with services via transactions, which contain a binder token, code of the method to execute, raw data buffer, and sender PID/UID (added by the driver)

- Most-low-level operations and data structures (i.e. `Parcel`) are abstracted by `libbinder` (at the native level), which is what the clients and services use.

- Except that clients and services don’t want to know anything about the Binder protocol and `libbinder`, so they make use of proxies and stubs:

![Diagram of Binder communication](image)

- Java-based proxies and stubs can be automatically generated by `aidl` tool for services described with AIDL.

- In fact, most clients don’t even want to know that they are using IPC, never mind Binder, or proxies, so they count on managers to abstract all of that complexity for them:
This is in particular true for system services, which typically expose only a subset of their APIs to the clients via their managers.

- But how does the client get a handle to the service it wants to talk to? Just ask the `servicemanager` (Binder's `CONTEXT_MGR`), and hope that the service has already registered with it:

For security/sanity reasons, the binder driver will only accept a single/one-time `CONTEXT_MGR` registration, which is why `servicemanager` is among the first services to start on Android.

To get a list of services currently registered with `servicemanager`, run:

```bash
$ adb shell service list
Found 71 services:
  0 sip: [android.net.sip.ISipService]
  1 phone: [com.android.internal.telephony.ITelephony]
    ... 20 location: [android.location.ILocationManager]
    ... 55 activity: [android.app.IActivityManager]
    56 package: [android.content.pm.IPackageManager]
    ... 67 SurfaceFlinger: [android.ui.ISurfaceComposer]
  68 media.camera: [android.hardware.ICameraService]
  69 media.player: [android.media.IMediaPlayerService]
  70 media.audio_flinger: [android.media.IAudioFlinger]
```
Another way to look at it:

Client Process

0) get service from a context
1) invoke aService.foo(someBar) defined by AIDL
2) marshal someBar to dataParcel
3) transact(aService, TRANSACTION.foo, dataParcel, replyParcel, ...)
4) submit transaction/data via a blocking ioctl call

15) update the reply parcel
16) unmarshal someResult from replyParcel
17) return someResult

ServiceProcess

-2) create a service
9) invoke this.foo(someBar) defined by AIDL
8) unmarshal someBar from dataParcel
7) handle transaction onTransact(TRANSACTION.foo, dataParcel, replyParcel, ...)
6) get the binder object corresponding to the transaction
5) wake up from a blocking ioctl call and get the transaction data
13) submit replyParcel via a ioctl
10) return someResult
11) marshal someResult to replyParcel

Kemel

/dev/binder
Location Service: An Example

Location on Android
AIDL

- Android Interface Definition Language is a Android-specific language for defining Binder-based service interfaces
- AIDL follows Java-like interface syntax and allows us to declare our "business" methods
- Each Binder-based service is defined in a separate .aidl file, typically named IFooService.aidl, and saved in the src/ directory

```java
package com.example.app;
import com.example.app.Bar;
interface IFooService {
    void save(inout Bar bar);
    Bar getById(int id);
    void delete(in Bar bar);
    List<Bar> getAll();
}
```

- The aidl build tool (part of Android SDK) is used to extract a real Java interface (along with a Stub providing Android’s android.os.IBinder) from each .aidl file and place it into our gen/ directory

```java
gen/com/example/app/IFooService.java
```
Eclipse ADT automatically calls `aidl` for each `.aidl` file that it finds in our `src/` directory

- AIDL supports the following types:
  - `null`
  - `boolean, boolean[], byte, byte[], char[], int, int[], long, long[], float, float[], double, double[]`
  - `java.lang.CharSequence, java.lang.String` (sent as UTF-16)
- `java.io.FileDescriptor` - transferred as a dup of the original file descriptor (points to the same underlying stream and position)
- `java.io.Serializable` - not efficient (too verbose)
- `java.util.Map<String, Object>` - of supported types (always reconstructed as `java.util.HashMap`)
- `android.os.Bundle` - a specialized Map-wrapper that only accepts AIDL-supported data types
- `java.util.List` - of supported types (always reconstructed as `java.util.ArrayList`)
- `java.lang.Object[]` - of supported types (including primitive wrappers)
- `android.util.SparseArray`, `android.util.SparseBooleanArray`  
- `android.os.IBinder`, `android.os.IInterface` - transferred by (globally unique) reference (as a "strong binder", a.k.a. handle) that can be used to call-back into the sender
- `android.os.Parcel` - allowing for custom types:

```java
package com.example.app;

import android.os.Parcel;
import android.os.Parcelable;

public class Bar implements Parcelable {
    private int id;
    private String data;

    public Bar(int id, String data) {
        this.id = id;
        this.data = data;
    }

    // getters and setters omitted
    ...
    public int describeContents() {
        return 0;
    }

    public void writeToParcel(Parcel parcel, int flags) {
        parcel.writeInt(this.id);
        parcel.writeString(this.data);
    }

    public void readFromParcel(Parcel parcel) {
        this.id = parcel.readInt();
        this.data = parcel.readString();
    }

    public static final Parcelable.Creator<Bar> CREATOR = new Parcelable.Creator<Bar>() {
        public BarcreateFromParcel(Parcel parcel) {
            return new Bar(parcel.readInt(), parcel.readString());
        }
        public Bar[] newArray(int size) {
            return new Bar[size];
        }
    };
```
Here, the `public void readFromParcel(Parcel) method is not defined by the Parcelable interface. Instead, it would be required here because Bar is considered mutable - i.e. we expect the remote side to be able to change it in `void save(inout Bar bar) method.

Similarly, `public static final Parcelable.Creator<Bar> CREATOR field is also not defined by the Parcelable interface (obviously). It reconstructs Bar from _data parcel in save transaction, and from _reply parcel in getById operation.

- These custom classes have to be declared in their own (simplified) `.aidl` files

```java
package com.example.app.Bar;
parcelable Bar;
```

AIDL-interfaces have to `import` parcelable custom classes even if they are in the same package. In the case of the previous example, `src/com/example/app/IFooService.aidl` would have to `import com.example.app.Bar;` if it makes any references to `com.example.app.Bar` even though they are in the same package.

- AIDL-defined methods can take zero or more parameters, and must return a value or `void`
- All non-primitive parameters require a `directional tag` indicating which way the data goes: one of: `in`, `out`, or `inout`
  - Direction for primitives is always `in` (can be omitted)
  - The direction tag tells binder when to marshal the data, so its use has direct consequences on performance
- All `.aidl` comments are copied over to the generated Java interface (except for comments before the import and package statements).
- Only the following exceptions are implicitly supported: `SecurityException`, `BadParcelableException`, `IllegalArgumentException`, `NullPointerException`, and `IllegalStateException`
- Static fields are not supported in `.aidl` files
Binder Object Reference Mapping Across Process Boundaries

- A binder object reference is one of the following
  - An actual virtual memory address to a binder object in the same process
  - An abstract 32-bit handle to a binder object in another process

- On every transaction, the binder driver automatically maps local addresses to remote binder handles and remote binder handles to local addresses

- This mapping is done on:
  - Targets of binder transactions
    - IBinder object references shared across process boundaries as a parameter or a return value (embedded in transaction data)

- For this to work
  - The driver maintains mappings of local addresses and remote handles between processes (as a binary-tree per process) so that it can perform this translation
  - References embedded in transaction data are discovered based on offsets that the client provides when it submits its transaction and then rewritten in-place

- The binder driver does not know anything about binder objects that have never been shared with a remote process
  - Once a new binder object reference is found in a transaction, it is remembered by binder
  - Any time that reference is shared with another process, its reference count is incremented
  - 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, and binder removes its mapping
To demonstrate a Binder-based service and client (based on Fibonacci), we’ll create three separate projects:

1. **FibonacciCommon** library project - to define our AIDL interface as well as custom types for parameters and return values

2. **FibonacciService** project - where we implement our AIDL interface and expose it to the clients

3. **FibonacciClient** project - where we connect to our AIDL-defined service and use it

The code is available
- As a ZIP archive: https://github.com/marakana/FibonacciBinderDemo/zipball/master
- By Git: `git clone` https://github.com/marakana/FibonacciBinderDemo.git

- The UI will roughly look like this when done:
**FibonacciCommon - Define AIDL Interface and Custom Types**

- We start by creating a new Android (library) project, which will host the common API files (an AIDL interface as well as custom types for parameters and return values) shared by the service and its clients
  - Project Name: FibonacciCommon
  - Build Target: Android 2.2 (API 8) or later
  - Package Name: com.marakana.android.fibonaccicommon
  - Min SDK Version: 8 or higher
  - No need to specify Application name or an activity

- To turn this into a library project we need to access project properties → Android → Library and check **Is Library**
  - We could also manually add android.library=true to FibonacciCommon/default.properties and refresh the project

- Since library projects are never turned into actual applications (APKs)
  - We can simplify our manifest file:
    ```xml
    <?xml version="1.0" encoding="utf-8"?>
    <manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.marakana.android.fibonaccicommon" android:versionCode="1"
    android:versionName="1.0">
    </manifest>
    
    - And we can remove everything from FibonacciCommon/res/ directory (e.g. rm -fr FibonacciCommon/res/*)

- We are now ready to create our AIDL interface
  FibonacciCommon/src/com/marakana/android/fibonaccicommon/IFibonacciService.aidl
  ```java
  package com.marakana.android.fibonaccicommon;
  
  import com.marakana.android.fibonaccicommon.FibonacciRequest;
  import com.marakana.android.fibonaccicommon.FibonacciResponse;
  
  interface IFibonacciService {
    long fibJR(in long n);
    long fibJI(in long n);
    long fibNR(in long n);
    long fibNI(in long n);
    FibonacciResponse fib(in FibonacciRequest request);
  }
  ```
package com.marakana.android.fibonaccicommon;

import android.os.Parcel;
import android.os.Parcelable;

public class FibonacciRequest implements Parcelable {
    public static enum Type {
        RECURSIVE_JAVA, ITERATIVE_JAVA, RECURSIVE_NATIVE, ITERATIVE_NATIVE
    }

    private final long n;
    private final Type type;

    public FibonacciRequest(long n, Type type) {
        this.n = n;
        if (type == null) {
            throw new NullPointerException("Type must not be null");
        }
        this.type = type;
    }

    public long getN() {
        return n;
    }

    public Type getType() {
        return type;
    }

    public int describeContents() {
        return 0;
    }

    public void writeToParcel(Parcel parcel, int flags) {
        parcel.writeLong(this.n);
        parcel.writeInt(this.type.ordinal());
    }

    public static final Parcelable.Creator<FibonacciRequest> CREATOR = new
    Parcelable.Creator<FibonacciRequest>() {
        public FibonacciRequest createFromParcel(Parcel in) {
            long n = in.readLong();
            Type type = Type.values()[in.readInt()];
            return new FibonacciRequest(n, type);
        }

        public FibonacciRequest[] newArray(int size) {
            return new FibonacciRequest[size];
        }
    };

FibonacciCommon/src/com/marakana/android/fibonaccicommon/FibonacciRequest.aidl
package com.marakana.android.fibonaccicommon;

parcelable FibonacciRequest;

FibonacciCommon/src/com/marakana/android/fibonaccicommon/FibonacciResponse.java

package com.marakana.android.fibonaccicommon;

import android.os.Parcel;
import android.os.Parcelable;

public class FibonacciResponse implements Parcelable {

    private final long result;
    private final long timeInMillis;

    public FibonacciResponse(long result, long timeInMillis) {
        this.result = result;
        this.timeInMillis = timeInMillis;
    }

    public long getResult() {
        return result;
    }

    public long getTimeInMillis() {
        return timeInMillis;
    }

    public int describeContents() {
        return 0;
    }

    public void writeToParcel(Parcel parcel, int flags) {
        parcel.writeLong(this.result);
        parcel.writeLong(this.timeInMillis);
    }

    public static final Parcelable.Creator<FibonacciResponse> CREATOR = new Parcelable.Creator<FibonacciResponse>() {
        public FibonacciResponse createFromParcel(Parcel in) {
            return new FibonacciResponse(in.readLong(), in.readLong());
        }

        public FibonacciResponse[] newArray(int size) {
            return new FibonacciResponse[size];
        }
    };

FibonacciCommon/src/com/marakana/android/fibonaccicommon/FibonacciResponse.aidl

package com.marakana.android.fibonaccicommon;

parcelable FibonacciResponse;

- Finally we are now ready to take a look at our generated Java interface

FibonacciCommon/gen/com/marakana/android/fibonaccicommon/IFibonacciService.java
package com.marakana.android.fibonaccicommon;

public interface IFibonacciService extends android.os.IInterface {
    
    public static abstract class Stub extends android.os.Binder {
        implements com.marakana.android.fibonacci.IFibonacciService {
            ...
            public static com.marakana.android.fibonacci.IFibonacciService asInterface(
                android.os.IBinder obj) {
                ...
            }
            public android.os.IBinder asBinder() {
                return this;
            }
            ...
        }
    }

    public long fibJR(long n) throws android.os.RemoteException;
    public long fibJI(long n) throws android.os.RemoteException;
    public long fibNR(long n) throws android.os.RemoteException;
    public long fibNI(long n) throws android.os.RemoteException;
    public com.marakana.android.fibonaccicommon.FibonacciResponse fib( 
        com.marakana.android.fibonaccicommon.FibonacciRequest request)
        throws android.os.RemoteException;
}
FibonacciService - Implement AIDL Interface and Expose It To Our Clients

- We start by creating a new Android project, which will host the our AIDL Service implementation as well as provide a mechanism to access (i.e. bind to) our service implementation
  - Project Name: FibonacciService
  - Build Target: Android 2.2 (API 8) or later
  - Package Name: com.marakana.android.fibonacciService
  - Application name: Fibonacci Service
  - Min SDK Version: 8 or higher
  - No need to specify an Android activity

- We need to link this project to the FibonacciCommon in order to be able to access the common APIs:
  - project properties → Android → Library → Add… → FibonacciCommon
  - As the result, FibonacciService/default.properties now has
    android.library.reference.1=../FibonacciCommon and
    FibonacciService/.classpath and FibonacciService/.project also link to
    FibonacciCommon

- Our service will make use of the com.marakana.android.fibonaccinative.FibLib, which provides the actual implementation of the Fibonacci algorithms

- We copy (or move) this Java class (as well as the jni/ implementation) from the FibonacciNative project
  - Don’t forget to run ndk-build under FibonacciService/ in order to generate the required
    native library

- We are now ready to implement our AIDL-defined interface by extending from the auto-generated
  com.marakana.android.fibonaccicommon.IFibonacciService.Stub (which in turn
  extends from android.os.Binder)
  FibonacciService/src/com/marakana/android/fibonacciService/IFibonacciServiceImpl.java
package com.marakana.android.fibonacciService;

import android.os.SystemClock;
import android.util.Log;

import com.marakana.android.fibonaccicommon.FibonacciRequest;
import com.marakana.android.fibonaccicommon.FibonacciResponse;
import com.marakana.android.fibonaccicommon.IFibonacciService;
import com.marakana.android.fibonaccicommon.FibonacciRequests;
import com.marakana.android.fibonaccicommon.FibonacciResponses;

public class IFibonacciServiceImpl extends IFibonacciService.Stub {
    private static final String TAG = "IFibonacciServiceImpl";

    public long fibJI(long n) {
        Log.d(TAG, String.format("fibJI(%d)", n));
        return FibLib.fibJI(n);
    }

    public long fibJR(long n) {
        Log.d(TAG, String.format("fibJR(%d)", n));
        return FibLib.fibJR(n);
    }

    public long fibNI(long n) {
        Log.d(TAG, String.format("fibNI(%d)", n));
        return FibLib.fibNI(n);
    }

    public long fibNR(long n) {
        Log.d(TAG, String.format("fibNR(%d)", n));
        return FibLib.fibNR(n);
    }

    public FibonacciResponse fib(FibonacciRequest request) {
        Log.d(TAG,
             String.format("fib(%d, %s)"%, request.getN(), request.getType()));
        long timeInMillis = SystemClock.uptimeMillis();
        long result;
        switch (request.getType()) {
            case ITERATIVE_JAVA:
                result = this.fibJI(request.getN());
                break;
            case RECURSIVE_JAVA:
                result = this.fibJR(request.getN());
                break;
            case ITERATIVE_NATIVE:
                result = this.fibNI(request.getN());
                break;
            case RECURSIVE_NATIVE:
                result = this.fibNR(request.getN());
                break;
            default:
                return null;
        }
        timeInMillis = SystemClock.uptimeMillis() - timeInMillis;
        return new FibonacciResponse(result, timeInMillis);
    }
}
Expose our AIDL-defined Service Implementation to Clients

- In order for clients (callers) to use our service, they first need to bind to it.
- But in order for them to bind to it, we first need to expose it via our own android.app.Service's onBind(Intent) implementation.

```java
package com.marakan.android.fibonacci.service;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;

public class FibonacciService extends Service { // 1

    private static final String TAG = "FibonacciService";

    private IFibonacciServiceImpl service; // 2

    @Override
    public void onCreate() {
        super.onCreate();
        this.service = new IFibonacciServiceImpl(); // 3
        Log.d(TAG, "onCreate() 'ed"); // 4
    }

    @Override
    public IBinder onBind(Intent intent) {
        Log.d(TAG, "onBind() 'ed"); // 5
        return this.service; // 6
    }

    @Override
    public boolean onUnbind(Intent intent) {
        Log.d(TAG, "onUnbind() 'ed"); // 4
        return super.onUnbind(intent);
    }

    @Override
    public void onDestroy() {
        Log.d(TAG, "onDestroy() 'ed");
        this.service = null;
        super.onDestroy();
    }
}
```

1. We create yet another "service" object by extending from android.app.Service. The purpose of FibonacciService object is to provide access to our Binder-based IFibonacciServiceImpl object.
2. Here we simply declare a local reference to IFibonacciServiceImpl, which will act as a singleton (i.e. all clients will share a single instance). Since our IFibonacciServiceImpl does
not require any special initialization, we could instantiate it at this point, but we choose to delay this until the `onCreate()` method.

Now we instantiate our `IFibonacciServiceImpl` that we'll be providing to our clients (in the `onBind(Intent)` method). If our `IFibonacciServiceImpl` required access to the `Context` (which it doesn't) we could pass a reference to `this` (i.e. `android.app.Service`, which implements `android.content.Context`) at this point. Many Binder-based services use `Context` in order to access other platform functionality.

This is where we provide access to our `IFibonacciServiceImpl` object to our clients. By design, we chose to have only one instance of `IFibonacciServiceImpl` (so all clients share it) but we could also provide each client with their own instance of `IFibonacciServiceImpl`.

Finally, we register our `FibonacciService` in our `AndroidManifest.xml`, so that clients can find it.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.marakana.android.fibonacci_service" android:versionCode="1"
    android:versionName="1.0">
    <uses-sdk android:minSdkVersion="8" />
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <service android:name=".FibonacciService">
            <intent-filter>
                <action android:name="com.marakana.android.fibonacci_common(IFibonacciService)" />
            </intent-filter>
        </service>
    </application>
</manifest>
```

The name of this action is arbitrary, but it is a common convention to use the fully-qualified name of our AIDL-derived interface.
FibonacciClient - Using AIDL-defined Binder-based Services

- We start by creating a new Android project, which will server as the client of the AIDL Service we previously implemented
  - Project Name: FibonacciClient
  - Build Target: Android 2.2 (API 8) or later
  - Package Name: com.marakana.android.fibonacciClient
  - Application name: Fibonacci Client
  - Create activity: FibonacciActivity
    - We’ll repurpose most of this activity’s code from FibonacciNative
  - Min SDK Version: 8 or higher

- We need to link this project to the FibonacciCommon in order to be able to access the common APIs:
  project properties → Android → Library → Add… → FibonacciCommon
  - As the result, FibonacciClient/default.properties now has
    android.library.reference.1=../FibonacciCommon and
    FibonacciClient/.classpath and FibonacciClient/.project also link to
    FibonacciCommon
  - As an alternative, we could’ve avoided creating FibonacciCommon in the first place
    - FibonacciService and FibonacciClient could have each had a copy of:
      IFibonacciService.aidl, FibonacciRequest.aidl, FibonacciResponse.aidl,
      FibonacciResult.java, and FibonacciResponse.java++
    - But we don’t like duplicating source code (even though the binaries do get duplicated at runtime)

- Our client will make use of the string resources and layout definition from FibonacciNative application
  FibonacciClient/res/values/strings.xml
We are now ready to implement our client

FibonacciClient/src/com/marakana/android/fibonacciClient/FibonacciActivity.java

```java
package com.marakana.android.fibonacciClient;

import android.app.Activity;
import android.app.ProgressDialog;
import android.content.ComponentName;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.AsyncTask;
```
import android.os.Bundle;
import android.os.IBinder;
import android.os.RemoteException;
import android.os.SystemClock;
import android.text.TextUtils;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioButton;
import android.widget.TextView;
import android.widget.Toast;

import com.marakana.android.fibonaccicommon.FibonacciRequest;
import com.marakana.android.fibonaccicommon.FibonacciResponse;
import com.marakana.android.fibonaccicommon.IFibonacciService;

public class FibonacciActivity extends Activity implements OnClickListener, ServiceConnection {

    private static final String TAG = "FibonacciActivity";

    private EditText input; // our input n
    private Button button; // trigger for fibonacci calculation
    private RadioGroup type; // fibonacci implementation type
    private TextView output; // destination for fibonacci result
    private IFibonacciService service; // reference to our service

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        super.setContentView(R.layout.main);
        // connect to our UI elements
        this.input = (EditText) super.findViewById(R.id.input);
        this.button = (Button) super.findViewById(R.id.button);
        this.type = (RadioGroup) super.findViewById(R.id.type);
        this.output = (TextView) super.findViewById(R.id.output);
        // request button click call-backs via OnClickListener method
        this.button.setOnClickListener(this);
        // the button will be enabled once we connect to the service
        this.button.setEnabled(false);
    }

    @Override
    protected void onResume() {
        Log.d(TAG, "onResume()"d);
        super.onResume();
        // Bind to our FibonacciService service, by looking it up by its name
        // and passing ourselves as the ServiceConnection object
        // We'll get the actual IFibonacciService via a callback to
        // onServiceConnected() below
        if (!super.bindService(new Intent(IFibonacciService.class.getName()),
                this, BIND_AUTO_CREATE)) {
            Log.w(TAG, "Failed to bind to service");
        }
    }
}
protected void onPause() {
    Log.d(TAG, "onPause()'ed");
    super.onPause();
    // No need to keep the service bound (and alive) any longer than
    // necessary
    super.unbindService(this);
}

public void onServiceConnected(ComponentName name, IBinder service) {
    Log.d(TAG, "onServiceConnected()'ed to " + name);
    // finally we can get to our IFibonacciService
    this.service = IFibonacciService.Stub.asInterface(service);
    // enable the button, because the IFibonacciService is initialized
    this.button.setEnabled(true);
}

public void onServiceDisconnected(ComponentName name) {
    Log.d(TAG, "onServiceDisconnected()'ed to " + name);
    // our IFibonacciService service is no longer connected
    this.service = null;
    // disabled the button, since we cannot use IFibonacciService
    this.button.setEnabled(false);
}

// handle button clicks
public void onClick(View view) {
    // parse n from input (or report errors)
    final long n;
    String s = this.input.getText().toString();
    if (TextUtils.isEmpty(s)) {
        return;
    }
    try {
        n = Long.parseLong(s);
    } catch (NumberFormatException e) {
        this.input.setError(super.getText(R.string.input_error));
        return;
    }

    // build the request object
    final FibonacciRequest.Type type;
    switch (FibonacciActivity.this.type.getCheckedRadioButtonId()) {
    case R.id.type_fib_jr:
        type = FibonacciRequest.Type.RECURSIVE_JAVA;
        break;
    case R.id.type_fib_ji:
        type = FibonacciRequest.Type.ITERATIVE_JAVA;
        break;
    case R.id.type_fib_nr:
        type = FibonacciRequest.Type.RECURSIVE_NATIVE;
        break;
    case R.id.type_fib_ni:
        type = FibonacciRequest.Type.ITERATIVE_NATIVE;
        break;
    default:
        return;
    }
    final FibonacciRequest request = new FibonacciRequest(n, type);

    // showing the user that the calculation is in progress
    final ProgressDialog dialog = ProgressDialog.show(this, ",",
super.getText(R.string.progress_text), true);
// since the calculation can take a long time, we do it in a separate
// thread to avoid blocking the UI
new AsyncTask_VOID, Void, String>() {
    @Override
    protected String doInBackground(Void... params) {
        // this method runs in a background thread
        try {
            long totalTime = SystemClock.uptimeMillis();
            FibonacciResponse response = FibonacciActivity.this.service
                .fib(request);
            totalTime = SystemClock.uptimeMillis() - totalTime;
            // generate the result
            return String.format(
                "fibonacci(%d)=%d\n\n%d ms\n+%d ms",
                n, response.getResult(), response.getTimeInMillis(),
                totalTime - response.getTimeInMillis());
        } catch (RemoteException e) {
            Log.wtf(TAG, "Failed to communicate with the service", e);
            return null;
        }
    }

    @Override
    protected void onPostExecute(String result) {
        // get rid of the dialog
dialog.dismiss();
        if (result == null) {
            // handle error
            Toast.makeText(FibonacciActivity.this, R.string.fib_error,
                Toast.LENGTH_SHORT).show();
        } else {
            // show the result to the user
            FibonacciActivity.this.output.setText(result);
        }
    }
}.execute(); // run our AsyncTask
}

We should avoid having an implicit (but strong) reference from our AsyncTask to our activity. Here we took a shortcut for brevity reasons.

- Our activity should already be registered in our AndroidManifest.xml file

FibonacciClient/AndroidManifest.xml
And the result should look like:

![Fibonacci Client](image)

Fibonacci Client

Get Your Fibonacci Here!

34

fibR fibJ fibNR fibNI

Get Fib Result

fibonacci(34)=5702887 in 4986 ms
Async Binder IPC (by Example)

- Binder allows for the asynchronous communication between the client and its service via the oneway declaration on the AIDL interface
- Of course, we still care about the result, so generally async calls are used with call-backs - typically through listeners
- When clients provide a reference to themselves as call-back listeners, then the roles reverse at the time the listeners are called: clients' listeners become the services, and services become the clients to those listeners
- This is best explained via an example (based on Fibonacci)
- The code is available
  - As a ZIP archive: https://github.com/marakana/FibonacciAsyncBinderDemo/zipball/master
  - By Git: git clone https://github.com/marakana/FibonacciAsyncBinderDemo.git
FibonacciCommon - Defining a oneway AIDL Service

- First, we need a listener, which itself is a oneway AIDL-defined "service":

```aidl
package com.marakan.android.fibonaccicommon;

import com.marakan.android.fibonaccicommon.FibonacciRequest;
import com.marakan.android.fibonaccicommon.FibonacciResponse;

oneway interface IFibonacciServiceResponseListener {
    void onResponse(in FibonacciResponse response);
}
```

- Now we can create our oneway (i.e. asynchronous) interface:

```aidl
package com.marakan.android.fibonaccicommon;

import com.marakan.android.fibonaccicommon.FibonacciRequest;
import com.marakan.android.fibonaccicommon.FibonacciResponse;
import com.marakan.android.fibonaccicommon.IFibonacciServiceResponseListener;

oneway interface IFibonacciService {
    void fib(in FibonacciRequest request, in IFibonacciServiceResponseListener listener);
}
```
FibonacciService - Implementing our async AIDL service

- The implementation of our service invokes the listener, as opposed to returning a result:

```java
package com.marakan.android.fibonacci.service;

import android.os.RemoteException;
import android.os.SystemClock;
import android.util.Log;

import com.marakan.android.fibonacci.common.FibonacciRequest;
import com.marakan.android.fibonacci.common.FibonacciResponse;
import com.marakan.android.fibonacci.common.IFibonacciService;
import com.marakan.android.fibonacci.common.FibonacciServiceResponseListener;
import com.marakan.android.fibonacci.common.FibLib;

public class IFibonacciServiceImpl extends IFibonacciService.Stub {
  private static final String TAG = "IFibonacciServiceImpl";

  @Override
  public void fib(FibonacciRequest request,
                  IFibonacciServiceResponseListener listener) throws RemoteException {
    long n = request.getN();
    Log.d(TAG, "fib( + n + ")");
    long timeInMillis = SystemClock.uptimeMillis();
    long result;
    switch (request.getType()) {
      case ITERATIVE_JAVA:
        result = FibLib.fibJI(n);
        break;
      case RECURSIVE_JAVA:
        result = FibLib.fibJR(n);
        break;
      case ITERATIVE_NATIVE:
        result = FibLib.fibIN(n);
        break;
      case RECURSIVE_NATIVE:
        result = FibLib.fibNR(n);
        break;
      default:
        result = 0;
    }
    timeInMillis = SystemClock.uptimeMillis() - timeInMillis;
    Log.d(TAG, String.format("Got fib(\d\d) = \d\d in %d ms", n, result, timeInMillis));
    listener.onResponse(new FibonacciResponse(result, timeInMillis));
  }
}
```

The service will not block waiting for the listener to return, because the listener itself is also oneway.
FibonacciClient - Implementing our async AIDL client

- Finally, we implement our client, which itself has to also implement a listener as a Binder service: FibonacciClient/src/com/marakana/android/fibonacciclient/FibonacciActivity.java:

```java
package com.marakana.android.fibonacciclient;

import android.app.Activity;
import android.app.Dialog;
import android.app.ProgressDialog;
import android.content.ComponentName;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.Handler;
import android.os.IBinder;
import android.os.Message;
import android.os.RemoteException;
import android.os.SystemClock;
import android.text.TextUtils;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.RadioButton;
import android.widget.TextView;

import com.marakana.android.fibonaccicommon.FibonacciRequest;
import com.marakana.android.fibonaccicommon.FibonacciResponse;
import com.marakana.android.fibonaccicommon.IFibonacciService;
import com.marakana.android.fibonaccicommon.IFibonacciServiceResponseListener;

public class FibonacciActivity extends Activity implements OnClickListener, ServiceConnection {

    private static final String TAG = "FibonacciActivity";

    // the id of a message to our response handler
    private static final int RESPONSE_MESSAGE_ID = 1;

    // the id of a progress dialog that we'll be creating
    private static final int PROGRESS_DIALOG_ID = 1;

    private EditText input; // our input n

    private Button button; // trigger for fibonacci calculation

    private RadioGroup type; // fibonacci implementation type

    private TextView output; // destination for fibonacci result

    private IFibonacciService service; // reference to our service

    // the responsibility of the responseHandler is to take messages
```
private final Handler responseHandler = new Handler() {
    @Override
    public void handleMessage(Message message) {
        switch (message.what) {
            case RESPONSE_MESSAGE_ID:
                Log.d(TAG, "Handling response");
                this.output.setText((String) message.obj);
                this.removeDialog(PROGRESS_DIALOG_ID);
                break;
            }
        }
    }
};

private final IFibonacciServiceResponseListener responseListener = new IFibonacciServiceResponseListener.Stub() {
    // this method is executed on one of the pooled binder threads
    @Override
    public void onResponse(FibonacciResponse response) throws RemoteException {
        String result = String.format("%d in %d ms", response.getResult(), response.getTimeInMillis());
        Log.d(TAG, "Got response: " + result);
        // since we cannot update the UI from a non-UI thread,
        // we'll send the result to the responseHandler (defined above)
        Message message = this.responseHandler
                .obtainMessage(RESPONSE_MESSAGE_ID, result);
        this.responseHandler.sendMessage(message);
    }
};

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    super.setContentView(R.layout.main);
    // connect to our UI elements
    this.input = (EditText) super.findViewById(R.id.input);
    this.button = (Button) super.findViewById(R.id.button);
    this.type = (RadioGroup) super.findViewById(R.id.type);
    this.output = (TextView) super.findViewById(R.id.output);
    // request button click call-backs via onClick(View) method
    this.button.setOnClickListener(this);
    // the button will be enabled once we connect to the service
    this.button.setEnabled(false);
}

@Override
protected void onStart() {
    Log.d(TAG, "onStart()\d");
    super.onStart();
    // Bind to our FibonacciService service, by looking it up by its name
    // and passing ourselves as the ServiceConnection object
    // We'll get the actual IFibonacciService via a callback to
    // onServiceConnected() below
    if (!super.bindService(new Intent(IFibonacciService.class.getName()),
                   this, BIND_AUTO_CREATE)) {
        Log.w(TAG, "Failed to bind to service");
    }
}
@Override
protected void onStop() {
    Log.d(TAG, "onStop()'ed");
    super.onStop();
    // No need to keep the service bound (and alive) any longer than
    // necessary
    super.unbindService(this);
}

public void onServiceConnected(ComponentName name, IBinder service) {
    Log.d(TAG, "onServiceConnected()'ed to " + name);
    // finally we can get to our IFibonacciService
    this.service = IFibonacciService.Stub.asInterface(service);
    // enable the button, because the IFibonacciService is initialized
    this.button.setEnabled(true);
}

public void onServiceDisconnected(ComponentName name) {
    Log.d(TAG, "onServiceDisconnected()'ed to " + name);
    // our IFibonacciService service is no longer connected
    this.service = null;
    // disabled the button, since we cannot use IFibonacciService
    this.button.setEnabled(false);
}

@Override
protected Dialog onCreateDialog(int id) {
    switch (id) {
    case PROGRESS_DIALOG_ID:
        // this dialog will be opened in onClick(...) and
        // dismissed/removed by responseHandler.handleMessage(...)
        ProgressDialog dialog = new ProgressDialog(this);
        dialog.setMessage(super.getText(R.string.progress_text));
        dialog.setIndeterminate(true);
        return dialog;
    default:
        return super.onCreateDialog(id);
    }
}

// handle button clicks
public void onClick(View view) {
    // parse n from input (or report errors)
    final long n;
    String s = this.input.getText().toString();
    if (TextUtils.isEmpty(s)) {
        return;
    }
    try {
        n = Long.parseLong(s);
    } catch (NumberFormatException e) {
        this.input.setError(super.getText(R.string.input_error));
        return;
    }

    // build the request object
    final FibonacciRequest.Type type;
    switch (FibonacciActivity.this.type.getCheckedRadioButtonId()) {
    case R.id.type_fib_jr:
        type = FibonacciRequest.Type.RECURSIVE_JAVA;
        break;
break;
case R.id.type_fib_ji:
    type = FibonacciRequest.Type.ITERATIVE_JAVA;
    break;
case R.id.type_fib_nr:
    type = FibonacciRequest.Type.RECURSIVE_NATIVE;
    break;
case R.id.type_fib_ni:
    type = FibonacciRequest.Type.ITERATIVE_NATIVE;
    break;
default:
    return;
}

final FibonacciRequest request = new FibonacciRequest(n, type);
try {
    Log.d(TAG, "Submitting request...");
    long time = SystemClock.uptimeMillis();
    // submit the request; the response will come to responseListener
    this.service.fib(request, this.responseListener);
    time = SystemClock.uptimeMillis() - time;
    Log.d(TAG, "Submitted request in " + time + " ms");
    // this dialog will be dismissed/removed by responseHandler
    super.showDialog(PROGRESS_DIALOG_ID);
} catch (RemoteException e) {
    Log.wtf(TAG, "Failed to communicate with the service", e);
}

Our listener should not retain a strong reference to the activity (and it does since it's an anonymous inner class), but in this case we skip on the correctness for the sake of brevity.
Sharing Memory via Binder

- Binder transactional data is copied among parties communicating - not ideal if we have a lot of data to send
  - In fact, binder imposes limits on how much data we can send via transactions
- If the data we want to share comes from a file, then we should just send the file descriptor instead
  - This is how we ask the media player to play an audio/video file for us - we just send it the FD
- If the data we want to send is located in memory, rather than trying to send all of it at once, we could send multiple but smaller chunks instead
  - Complicates our design
- Alternatively, we could take advantage of Android’s ashmem (Anonymous Shared Memory) facilities
  - Its Java wrapper android.os.MemoryFile is not meant for memory sharing from 3rd party apps
  - Drop to native (via JNI) and use ashmem directly?
- Native memory sharing implemented via frameworks/base/libs/binder/Parcel.cpp’s:
  - void Parcel::writeBlob(size_t len, WritableBlob* outBlob)
  - status_t Parcel::readBlob(size_t len, ReadableBlob* outBlob)
- This is roughly implemented as follows:

```cpp
Client
size_t len = 4096;
int fd = ashmem_create_region("Parcel Blob", len);
ashmem_set_prot_region(fd, PROT_READ | PROT_WRITE);
void* ptr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
ashmem_set_prot_region(fd, PROT_READ);
writeFileDescriptor(fd, true);
// write into ptr for len as desired
...
munmap(ptr, len);
close(fd);
```

```cpp
Service
int fd = readFileDescriptor();
void* ptr = mmap(NULL, len, PROT_READ, MAP_SHARED, fd, 0);
// read from ptr up to len as desired
...
munmap(ptr, len);
```
Removed error handling for brevity. Also, `writeFileDescriptor(...)` and `readFileDescriptor(...)` are provided by libbinder.
Limitations of Binder

- Binder supports a maximum of 15 binder threads per process

```cpp
static int open_driver()
{
    int fd = open("/dev/binder", O_RDWR);
    if (fd >= 0) {
        ...
        size_t maxThreads = 15;
        result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
        ...
    } else {
        ...
    }
    return fd;
}
...
```

- Avoid blocking binder threads
- If we need to perform a long-running task, it's better to spawn our own thread

- Binder limits its transactional buffer to 1Mb per process across all concurrent transactions
  - If arguments/return values are too large to fit into this buffer, TransactionTooLargeException is thrown
  - Because this buffer is shared across all transactions in a given process, many moderately sized transactions could also exhaust its limit
  - When this exception is thrown, we don't know whether we failed to send the request, or failed to receive the response
  - Keep transaction data small or use shared memory (ashmem)
Binder - Security

- Binder does directly deal with "security" concerns, but it enables a "trusted" execution environment and DAC.

- The binder driver allows only a single CONTEXT_MGR (i.e. servicemanager) to register:

```c
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    ...
    switch (cmd) {
    ...
    case BINDER_SET_CONTEXT_MGR:
        if (binder_context_mgr_node != NULL) {
            printk(KERN_ERR "binder: BINDER_SET_CONTEXT_MGR already set\n");
            ret = -EBUSY;
            goto err;
        }
        ...
        binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
        ...
    }
    ...
    ...
```

- The servicemanager in turn only allows registrations from trusted UIDs (like system, radio, media, etc.):

```c
frameworks/base/cmds/servicemanager/service_manager.c:
```
Each binder transaction carries in it the UID and PID of the sender, which we can easily access:

- `android.os.Binder.getCallingPid()`
- `android.os.Binder.getCallingUid()`
• Once we have the knowledge of the calling UID, we can easily resolve it the calling app via
  `PackageManager.get Packages ForUid(int uid)`
• Once we have the knowledge of the calling app, we can easily check whether it holds a permission we
  want to enforce via  `PackageManager.get Package Info(String packageName, int flags)`
  (with the `PackageManager.PERMISSION_GRANTED` flag)
• But, much easier to do permission enforcement via:
  1. `Context.checkCallingOrSelfPermission(String permission)` which returns
     `PackageManager.PERMISSION_GRANTED` if the calling process has been granted the
     permission or `PackageManager.PERMISSION_DENIED` otherwise
  2. `Context.enforceCallingPermission(String permission, String message)` - to
     automatically throw `SecurityException` if the caller does not have the requested permission

• This is how many of the application framework services enforce their permissions

![Diagram of Android permission flow]

• For example:
  `frameworks/base/services/java/com/android/server/VibratorService.java`
package com.android.server;
...
public class VibratorService extends IVibratorService.Stub {
    ...
    public void vibrate(long milliseconds, IBinder token) {
        if (mContext.checkSelfPermission(android.Manifest.permission.VIBRATE)
            != PackageManager.PERMISSION_GRANTED) {
            throw new SecurityException("Requires VIBRATE permission");
        }
        ...
    }
    ...
}

frameworks/base/services/java/com/android/server/LocationManagerService.java:
package com.android.server;
...
public class LocationManagerService extends ILocationManager.Stub implements Runnable {
    ...
    private static final String ACCESS_FINE_LOCATION =
        android.Manifest.permission.ACCESS_FINE_LOCATION;
    private static final String ACCESS_COARSE_LOCATION =
        android.Manifest.permission.ACCESS_COARSE_LOCATION;
    ...
    private void checkPermissionsSafe(String provider) {
        if (((LocationManager.GPS_PROVIDER.equals(provider))
            || LocationManager.PASSIVE_PROVIDER.equals(provider))
            && (mContext.checkSelfPermission(ACCESS_FINE_LOCATION)
            != PackageManager.PERMISSION_GRANTED)) {
            throw new SecurityException("Provider " + provider
                + " requires ACCESS_FINE_LOCATION permission");
        }
        if (LocationManager.NETWORK_PROVIDER.equals(provider)
            && (mContext.checkSelfPermission(ACCESS_FINE_LOCATION)
            != PackageManager.PERMISSION_GRANTED)
            && (mContext.checkSelfPermission(ACCESS_COARSE_LOCATION)
            != PackageManager.PERMISSION_GRANTED)) {
            throw new SecurityException("Provider " + provider
                + " requires ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION permission");
        }
    }
    ...
    private Location _getLastKnownLocationLocked(String provider) {
        checkPermissionsSafe(provider);
        ...
    }
    ...
    public Location getLastKnownLocation(String provider) {
        ...
        _getLastKnownLocationLocked(provider);
        ...
    }
}
Binder Death Notification

Given a reference to an **IBinder** object, we can:

- Ask whether the remote object is alive, via **Binder.isBinderAlive()** and **Binder.pingBinder()**
- Ask to be notified of its death, via **Binder.linkToDeath(IBinder.DeathRecipient recipient, int flags)**
  - This is useful if we want to clean up resources associated with a remote binder object (like a listener) that is no longer available
  - For example:

```java
public class LocationManagerService extends ILocationManager.Stub implements Runnable {
    ...
    private Receiver getReceiver(ILocationListener listener) {
        IBinder binder = listener.asBinder();
        Receiver receiver = mReceivers.get(binder);
        if (receiver == null) {
            receiver = new Receiver(listener);
            ...
            receiver.getListener().asBinder().linkToDeath(receiver, 0);
        }
        return receiver;
    }

    private final class Receiver implements IBinder.DeathRecipient, PendingIntent.OnFinished {
        final ILocationListener mListener;
        ...
        Receiver(ILocationListener listener) {
            mListener = listener;
            ...
        }
        ...
        public void binderDied() {
            ...
            removeUpdatesLocked(this);
        }
    }
    ...
}
```

---

**Note:** The code snippet provided is a simplified example and may not be directly applicable in all contexts. It illustrates the use of `IBinder` and `DeathRecipient` in Java, specifically in the context of Android's `LocationManagerService`. The actual implementation might vary depending on the specific requirements and context of the application.
Binder Reporting

Binder driver reports various stats on active/failed transactions via `/proc/binder/

- `/proc/binder/failed_transaction_log`
- `/proc/binder/state`
- `/proc/binder/stats`
- `/proc/binder/transaction_log`
- `/proc/binder/transactions`
- `/proc/binder/proc/<pid>`

Replace `/proc/binder` with `/sys/kernel/debug/binder` on devices with debugfs enabled.
Additional Binder Resources

- *Android Binder* by Thorsten Schreiber from Ruhr-Universität Bochum
- Android Binder IPC Mechanism - 0xLab by Jim Huang (黄敬群) from 0xlab
- Android’s Binder by Ken from Ken’s Space
- Dianne Hackborn on Binder in Android on Linux Kernel Mailing List archive (LKML.ORG)
- Android Binder on elinux.org
- Share memory using ashmem and binder in the android framework
- Introduction to OpenBinder and Interview with Dianne Hackborn
- Open Binder Documentation
- Binder IPC - A walk-through native `IAudioFlinger::setMode` call
Summary

We learned about:

- Why Android needs IPC
- What is Binder and how it differs from other forms of IPC
- Binder vs Intent/ContentProvider/Messenger-based IPC
- Binder Terminology
- Binder Communication and Discovery Model
- AIDL
- Binder Object Reference Mapping
- Synchronous vs Async Binder Invocations
- Memory Sharing
- Binder Limitations
- Security Implications
- Death Notification
- Reporting
Questions?

Didn’t we run out of time by now? :-)  
Thank you for your patience!  
Slides and screencast from this class will be posted to: http://mrkn.co/bgnhg
You can follow me here:

- @agargenta
- +Aleksandar Gargenta
- http://marakana.com/s/author/1/aleksandar_gargenta

This slide-deck is licensed under Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.