IPC communication

From Basic to not basic

IPC vs Thread

- Use Android IPC when you need to communicate between components in different processes. For example, you might use IPC to send a message to a service that is running in the background.
- Use Android threads when you need to execute multiple tasks concurrently within the same process. For example, you might use threads to update the UI while a long-running calculation is being performed.

Feature	**Android IPC**	**Android threads**
Purpose	Communication between components in different processes	Concurrent execution of tasks within the same process
Use cases	Sending messages to services, fetching data from content providers	Updating UI elements, performing long-running calculations, IO operation such as read/write file, database…

Different package will run in different process, is this 100% truth?

No

Can an APP has different processes running in a same package?

Yes

Type of process in Android APP

- By default, different packages will run in different processes in Android.

There are a few ways to allow different packages to run in the same process:

- Set the `android:process` attribute to the same value in the manifest files of the applications.
- Sign the applications with the same certificate.

How to configure - same package run in different process

```xml
<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

    package="com.example.myapp">

    <application android:process="process1">

        <service android:name=".ServerService" android:process="process2"/>

        <activity android:name=".FirstActivity" android:taskAffinity="process1">

        <activity android:name=".SecondActivity" android:taskAffinity="process2">

    </application>

</manifest>
```

How to configure - different package run in same process

com.example.myapp1

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp1">
    <application android:sharedUserId="process1">
        <!-- Activities and services in application 1 -->
    </application>
</manifest>
```

com.example.myapp2

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp2">
    <application android:sharedUserId="process1">
        <!-- Activities and services in application 2 -->
    </application>
</manifest>
```

Type of IPC communication in Android

Intents: Intents are a messaging mechanism used to communicate between components such as activities, services, and broadcast receivers. They can be used to start activities, start services, and send broadcast messages.

- **Bundles**: Bundles are used to pass data between activities or services using key-value pairs. They are often used in conjunction with intents to pass data along with the intent.

Content Providers: Content Providers allow different applications to share data with each other. They provide a standardized way to access and manipulate data stored in a central repository.

Broadcast Receivers: Broadcast Receivers allow applications to receive and respond to system-wide broadcast messages. They can be used to send and receive messages between different applications.

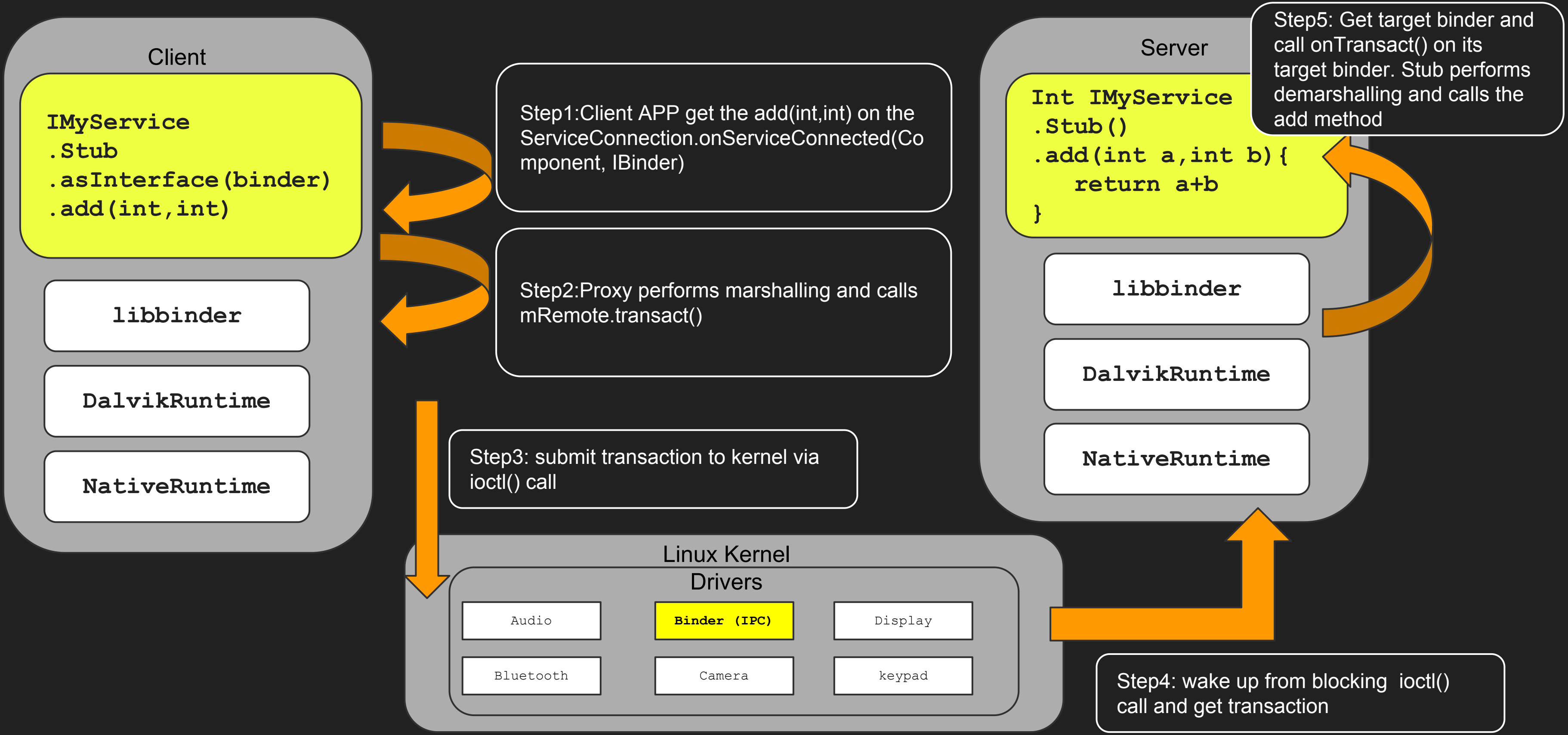
Type of IPC communication in Android

AIDL (Android Interface Definition Language): AIDL is a language used to define the interface between client and service in a remote process. It allows for communication between different applications running on separate processes.

Messenger: Messenger is a lightweight IPC mechanism that uses a message queue for communication between processes. It is based on the AIDL mechanism and provides a simpler interface for sending and receiving messages.

Local Broadcast Manager: Local Broadcast Manager is a mechanism for sending and receiving broadcast messages within the same application. It is more efficient than system-wide broadcasts as it does not involve inter-process communication.

AIDL - Android Interface Definition Language



Step1:Client APP get the add(int,int) on the ServiceConnection.onServiceConnected(Component, IBinder)

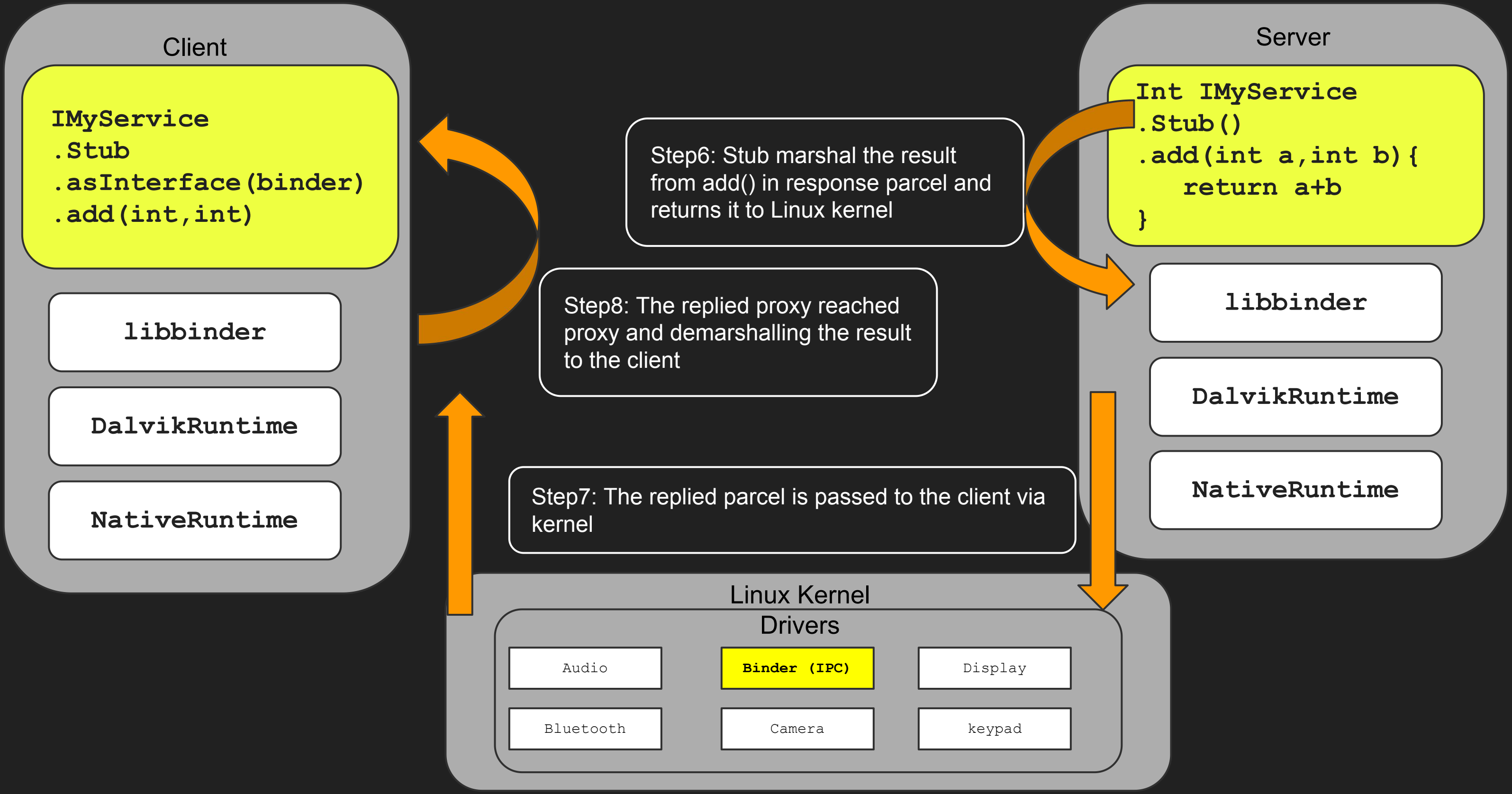
Step2:Proxy performs marshalling and calls mRemote.transact()

Step3: submit transaction to kernel via ioctl() call

Step4: wake up from blocking ioctl() call and get transaction

Step5: Get target binder and call onTransact() on its target binder. Stub performs demarshalling and calls the add method

AIDL - Android Interface Definition Language

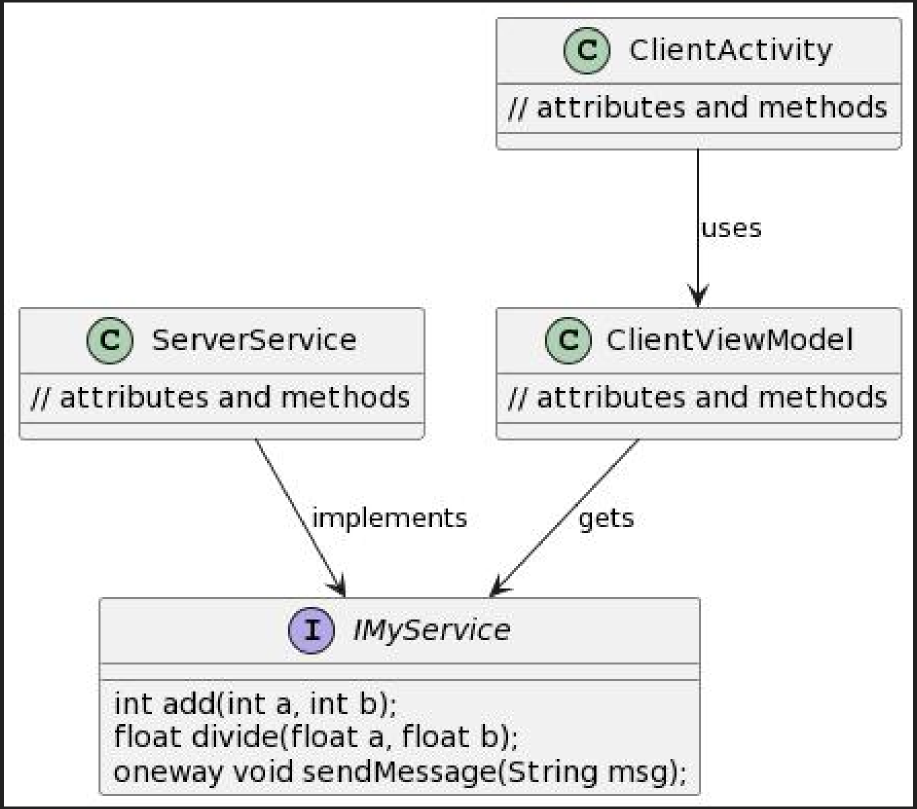


Step6: Stub marshal the result from add() in response parcel and returns it to Linux kernel

Step8: The replied proxy reached proxy and demarshalling the result to the client

Step7: The replied parcel is passed to the client via kernel

Project structure


ClientActivity
// attributes and methods
uses
ServerService
// attributes and methods
ClientViewModel
// attributes and methods
implements
gets
IMyService

```
int add(int a, int b);
float divide(float a, float b);
oneway void sendMessage(String msg);
```

ClientActivity uses ClientViewModel:
ClientActivity **depends** on ClientViewModel to perform certain operations or access data.
ClientActivity may call methods or access properties of ClientViewModel to update the UI or handle user interactions.

ClientViewModel gets IMyService:
ClientViewModel **has a** reference to IMyService to interact with the service and perform operations.
ClientViewModel calls methods of IMyService to retrieve data, send requests, or perform other actions related to the service.

ServerService implements IMyService:
ServerService **provides the implementation** for the functionality defined by the IMyService interface.
ServerService defines the methods and behavior required by the IMyService interface and handles the service-related operations.

Build gradle

To enable aidl in your module, you are required to explicitly enable it in build.gradle

```
buildFeatures { aidl = true }
```

How to create AIDL

- AIDL only accepts primitive type as parameters and return value
- AIDL only supports JAVA
- Complex data structure requires parcelable or serializable class
- Does not support method overloading, generic methods
- Like parcelable, AIDL interface orders and interface method needs to be well maintain to avoid any backward compatibility issue.
- The `oneway` keyword in AIDL is used to indicate that a method call does not require a response from the remote service. This can avoid blocking of ioctl call and improve overall performance

```
// aidl/com/example/myapp/IMyService.aidl
package com.example.myapp

interface IMyService {
    int add(int a, int b);
    float divide(float a, float b);
    oneway void sendMessage(String msg);
}
```

Implementation of the server side

```kotlin
// Service.kt
package com.example.myapp

import android.app.Service
import android.content.Intent
import android.os.IBinder

class MyService : Service() {
  private val mBinder: IMyService.Stub = object : IMyService.Stub() {
    override fun add(a: Int, b: Int): Int {
      return a + b
    }
    override fun divide(a: Float, b: Float): Float {
      return a / b
    }
    override fun sendMessage(message: String) {
      // do something with message
    }
  }

  override fun onBind(intent: Intent?): IBinder? {
    return mBinder
  }
}
```

Implement the AIDL interface

Provide the interface

In the client manifest

- Client manifest requires the BIND_REMOTE_SERVICE permission and enable the access to external package using the <queries>.

```
<uses-permission
android:name="android.permission.BIND_REMOTE_SERVICE" />

<queries>

   <package android:name="com.ressphere.myaidl" />

</queries>
```

```kotlin
// Client.kt
package com.example.myapp
import android.content.ComponentName
import android.content.Intent
import android.content.ServiceConnection
import android.os.IBinder
import android.os.RemoteException
import android.util.Log

class MyClient {
  private var mService: IMyService? = null
  private val mConnection: ServiceConnection = object : ServiceConnection {
    override fun onServiceConnected(name: ComponentName?, service: IBinder?) {
      mService = IMyService.Stub.asInterface(service)    Get interface
    }
    override fun onServiceDisconnected(name: ComponentName?) {
      mService = null
    }
  }

  fun add() {
    try {                                         Calling remote method via interface
      val result = mService?.add(1, 2)
      result?.let {Log.i("MyClient", "The result is: $it")}
    } catch (e: RemoteException) {
      Log.e("MyClient", "Failed to call add() on service", e)
    }
  }

  fun bind() {
    bindService(ComponentName("com.example.myapp", "com.example.myapp.MyService"), mConnection, BIND_AUTO_CREATE)
  }

  fun unbind() {
    unbindService(mConnection)
  }
}
```

How to pass complicated data between AIDL

To pass complicated data between AIDL (Android Interface Definition Language) components, you can use Parcelable objects. Parcelable is an interface that allows you to serialize and deserialize objects for efficient IPC (Inter-Process Communication) communication.

Example of parcelable class

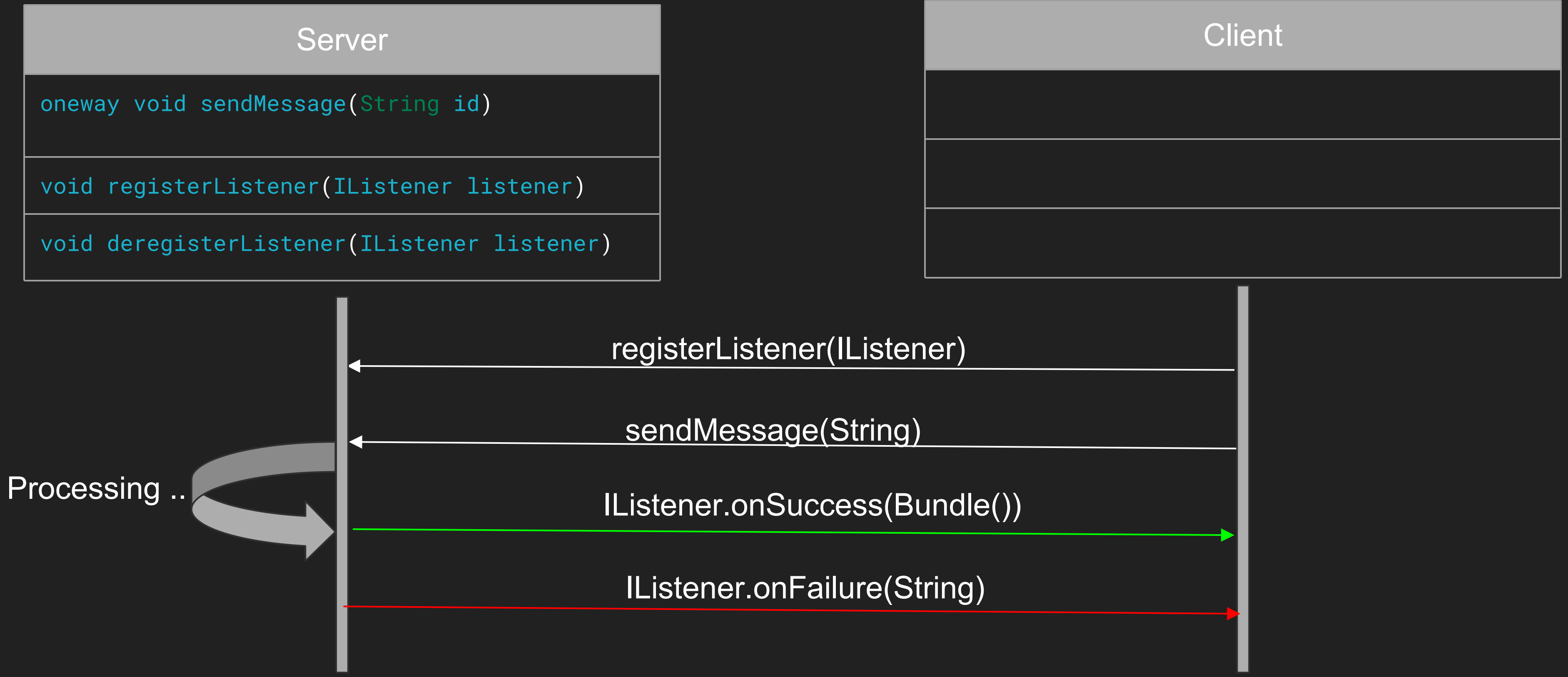
```kotlin
import android.os.Parcel
import android.os.Parcelable

data class MyData(val name: String, val age: Int) : Parcelable {
    constructor(parcel: Parcel) : this(parcel.readString() ?: "",
                                       parcel.readInt())

    override fun writeToParcel(parcel: Parcel, flags: Int) {
        parcel.writeString(name)
        parcel.writeInt(age)
     }

    override fun describeContents(): Int { return 0 }

    companion object CREATOR : Parcelable.Creator<MyData> {
        override fun createFromParcel(parcel: Parcel): MyData {return MyData(parcel)}
        override fun newArray(size: Int): Array<MyData?> {return arrayOfNulls(size)}
    }
}
```

Example of parcelize class

```
apply plugin: 'kotlin-android'
apply plugin: 'kotlinx-parcelize

android {
    // ...
}

dependencies {
    // ...
}
```

```
import android.os.Parcelable
import kotlinx.parcelize.Parcelize


@Parcelize
data class MyData(val name: String, val age: Int) : Parcelable
```

Create a Push Model in AIDL



Server

```
oneway void sendMessage(String id)
```

```
void registerListener(IListener listener)
```

```
void deregisterListener(IListener listener)
```

Client
registerListener(IListener)
sendMessage(String)
Processing ..
IListener.onSuccess(Bundle())
IListener.onFailure(String)

You can pass aidl interface as a parameter

```
// IListener.aidl
package com.ressphere.common;

interface IListener {
   void onSuccess(in Bundle result);
   void onFailure(String reason);
}
```

```
// IMyService.aidl
package com.ressphere.common;

import com.ressphere.common.IListener;
// Declare any non-default types here with import statements

interface IMyService {
   int add(int a, int b);
   float divide(float a, float b);
   oneway void sendMessage(String id);
   void registerListener(IListener listener);
   void deregisterListener(IListener listener);
}
```

Implement the new aidl function in Server

```
override fun sendMessage(id: String?) {
   val myData = MyData("anonymous", 100)
   val result = Bundle().also {
       it.putParcelable(id, myData)
   }
   val deadListeners = mutableListOf<IListener>()
   for(listener in listeners) {
       try {
           listener.onSuccess(result)
       } catch(deadObject: DeadObjectException) {
           Log.w(TAG, "listener is dead!!!")
           deadListeners.add(listener)
       }
   }
   listeners.removeAll(deadListeners)
}

override fun registerListener(listener: IListener) {
   listeners.add(listener)
}

override fun deregisterListener(listener: IListener) {
   listeners.remove(listener)
}
```

Register the listener in Client

```kotlin
private val listener = object : IListener.Stub() {
   override fun asBinder(): IBinder {
       return this
   }
   @RequiresApi(Build.VERSION_CODES.TIRAMISU)
   override fun onSuccess(result: Bundle?) {
       val data = result?.getParcelable("result", MyData::class.java)
       data?.let {_successResult.value = data.copy()}
   }
   override fun onFailure(reason: String?) {
       Log.e(TAG, "reason: $reason")
   }
}

private val serviceConnection = object : ServiceConnection {
   override fun onServiceConnected(name: ComponentName?,
                                   service: IBinder?) {
       Log.d(TAG, "onServiceConnected: $name")
       mService = IMyService.Stub.asInterface(service)
       mService?.registerListener(listener)
   }
   override fun onServiceDisconnected(name: ComponentName?) {
       mService?.deregisterListener(listener)
       mService = null
   }
}
```

Thank you

HIGH FIVE