Android Interface Definition Language (AIDL)

The Android Interface Definition Language (AIDL) is similar to other IDLs: it lets you define the programming interface that both the client and service agree upon in order to communicate with each other using interprocess communication (IPC).

On Android, one process can't normally access the memory of another process. To talk, they need to decompose their objects into primitives that the operating system can understand and marshall the objects across that boundary for you. The code to do that marshalling is tedious to write, so Android handles it for you with AIDL.

Note: AIDL is necessary only if you let clients from different applications access your service for IPC and you want to handle multithreading in your service. If you don't need to perform concurrent IPC across different applications, create your interface by implementing a `Binder`. If you want to perform IPC but *don't* need to handle multithreading, implement your interface using a `Messenger`. Regardless, be sure that you understand bound services before implementing an AIDL.

Before you begin designing your AIDL interface, be aware that calls to an AIDL interface are direct function calls. Don't make assumptions about the thread in which the call occurs. What happens is different depending on whether the call is from a thread in the local process or a remote process:

- Calls made from the local process execute in the same thread that is making the call. If this is your main UI thread, that thread continues to execute in the AIDL interface. If it is another thread, that is the one that executes your code in the service. Thus, if only local threads are accessing the service, you can completely control which threads are executing in it. But if that is the case, don't use AIDL at all; instead, create the interface by implementing a `Binder`.
- Calls from a remote process are dispatched from a thread pool the platform maintains inside your own process. Be prepared for incoming calls from unknown threads, with multiple calls happening at the same time. In other words, an implementation of an AIDL interface must be completely thread-safe. Calls made from one thread on the same remote object arrive *in order* on the receiver end.
- The `oneway` keyword modifies the behavior of remote calls. When it is used, a remote call does not block. It sends the transaction data and immediately returns. The implementation of the interface eventually receives this as a regular call from the `Binder` thread pool as a normal remote call. If `oneway` is used with a local call, there is no impact, and the call is still synchronous.

Defining an AIDL interface

Define your AIDL interface in an `.aidl` file using the Java programming language syntax, then save it in the source code, in the `src/` directory, of both the application hosting the service and any other application that binds to the service.

When you build each application that contains the `.aidl` file, the Android SDK tools generate an `IBinder` interface based on the `.aidl` file and save it in the project's `gen/` directory. The service must implement the `IBinder` interface as appropriate. The client applications can then bind to the service and call methods from the `IBinder` to perform IPC.

To create a bounded service using AIDL, follow these steps, which are described in the sections that follow:

1. Create the `.aidl` file
   This file defines the programming interface with method signatures.

2. Implement the interface
   The Android SDK tools generate an interface in the Java programming language based on your `.aidl` file. This interface has an inner abstract class named `Stub` that extends `Binder` and implements methods from your AIDL interface. You must extend the `Stub` class and implement the methods.

3. Expose the interface to clients
   Implement a `Service` and override `onBind()` to return your implementation of the `Stub` class.

Caution: Any changes that you make to your AIDL interface after your first release must remain backward compatible to avoid breaking other applications that use your service. That is, because your `.aidl` file must be copied to other applications so they can access your service's interface, you must maintain support for the original interface.

Create the .aidl file

AIDL uses a simple syntax that lets you declare an interface with one or more methods that can take parameters and return values. The parameters and return values can be of any type, even other AIDL-generated interfaces.

You must construct the `.aidl` file using the Java programming language. Each `.aidl` file must define a single interface and requires only the interface declaration and method signatures.

By default, AIDL supports the following data types:

- All primitive types in the Java programming language (such as `int`, `long`, `char`, `boolean`, and so on)
- Arrays of primitive types, such as `int[]`
- `String`
- `CharSequence`
- `List`
  All elements in the `List` must be one of the supported data types in this list or one of the other AIDL-generated interfaces or parcelables you declare. A `List` can optionally be used as a parameterized type class, such as `List<String>`. The actual concrete class that the other side receives is always an `ArrayList`, although the method is generated to use the `List` interface.

- `Map`
  All elements in the `Map` must be one of the supported data types in this list or one of the other AIDL-generated interfaces or parcelables you declare. Parameterized type maps, such as those of the form `Map<String,Integer>`, aren't supported. The actual concrete class that the other side receives is always a `HashMap`, although the method is generated to use the `Map` interface. Consider using a `Bundle` as an alternative to `Map`.

You must include an `import` statement for each additional type not listed previously, even if they are defined in the same package as your interface.

When defining your service interface, be aware that:

- Methods can take zero or more parameters and can return a value or void.

- All non-primitive parameters require a directional tag indicating which way the data goes: `in`, `out`, or `inout` (see the example below).
  Primitives, `String`, `IBinder`, and AIDL-generated interfaces are `in` by default and can't be otherwise.

  Caution: Limit the direction to what is truly needed, because marshalling parameters is expensive.

- All code comments included in the `.aidl` file are included in the generated `IBinder` interface except comments before the import and package statements.
- String and int constants can be defined in the AIDL interface, such as `const int VERSION = 1;`.
- Method calls are dispatched by a `transact()` code, which normally is based on a method index in the interface. Because this makes versioning difficult, you can manually assign the transaction code to a method: `void method() = 10;`.
- Nullable arguments and return types must be annotated using `@nullable`.

Here is an example `.aidl` file:

```
// IRemoteService.aidl
package com.example.android;

// Declare any non-default types here with import statements.

/** Example service interface */
interface IRemoteService {
    /** Request the process ID of this service. */
    int getPid();

    /** Demonstrates some basic types that you can use as parameters
     * and return values in AIDL.
     */
    void basicTypes(int anInt, long aLong, boolean aBoolean, float aFloat,
            double aDouble, String aString);
}
```

Save your `.aidl` file in your project's `src/` directory. When you build your application, the SDK tools generate the `IBinder` interface file in your project's `gen/` directory. The generated file's name matches the `.aidl` file's name, but with a `.java` extension. For example, `IRemoteService.aidl` results in `IRemoteService.java`.

If you use Android Studio, the incremental build generates the binder class almost immediately. If you do not use Android Studio, the Gradle tool generates the binder class next time you build your application. Build your project with `gradle assembleDebug` or `gradle assembleRelease` as soon as you finish writing the `.aidl` file, so that your code can link against the generated class.

Implement the interface

When you build your application, the Android SDK tools generate a `.java` interface file named after your `.aidl` file. The generated interface includes a subclass named `Stub` that is an abstract implementation of its parent interface, such as `YourInterface.Stub`, and declares all the methods from the `.aidl` file.

Note: `Stub` also defines a few helper methods, most notably `asInterface()`, which takes an `IBinder`, usually the one passed to a client's `onServiceConnected()` callback method, and returns an instance of the stub interface. For more details about how to make this cast, see the section Calling an IPC method.

To implement the interface generated from the `.aidl`, extend the generated `Binder` interface, such as `YourInterface.Stub`, and implement the methods inherited from the `.aidl` file.

Here is an example implementation of an interface called `IRemoteService`, defined by the preceding `IRemoteService.aidl` example, using an anonymous instance:

Kotlin Java

```
private val binder = object : IRemoteService.Stub() {
    override fun getPid(): Int =
            Process.myPid()
    override fun basicTypes(
            anInt: Int,
            aLong: Long,
            aBoolean: Boolean,
            aFloat: Float,
            aDouble: Double,
            aString: String
    ) {
        // Does nothing.
    }
}
```

Now the `binder` is an instance of the `Stub` class (a `Binder`), which defines the IPC interface for the service. In the next step, this instance is exposed to clients so they can interact with the service.

Be aware of a few rules when implementing your AIDL interface:

- Incoming calls are not guaranteed to execute on the main thread, so you need to think about multithreading from the start and properly build your service to be thread-safe.
- By default, IPC calls are synchronous. If you know that the service takes more than a few milliseconds to complete a request, don't call it from the activity's main thread. It might hang the application, resulting in Android displaying an "Application is Not Responding" dialog. Call it from a separate thread in the client.
- Only the exception types listed under the reference documentation for `Parcel.writeException()` are sent back to the caller.

Expose the interface to clients

Once you've implemented the interface for your service, you need to expose it to clients so they can bind to it. To expose the interface for your service, extend `Service` and implement `onBind()` to return an instance of your class that implements the generated `Stub`, as discussed in the preceding section. Here's an example service that exposes the `IRemoteService` example interface to clients.

Kotlin Java

```
class RemoteService : Service() {
    override fun onCreate() {
        super.onCreate()
    }
    override fun onBind(intent: Intent): IBinder {
        // Return the interface.
        return binder
    }
    private val binder = object : IRemoteService.Stub() {
        override fun getPid(): Int {
            return Process.myPid()
        }
        override fun basicTypes(
                anInt: Int,
                aLong: Long,
                aBoolean: Boolean,
                aFloat: Float,
                aDouble: Double,
                aString: String
        ) {
            // Does nothing.
        }
    }
}
```

Now, when a client, such as an activity, calls `bindService()` to connect to this service, the client's `onServiceConnected()` callback receives the `binder` instance returned by the service's `onBind()` method.

The client must also have access to the interface class. So if the client and service are in separate applications, then the client's application must have a copy of the `.aidl` file in its `src/` directory, which generates the `android.os.Binder` interface, providing the client access to the AIDL methods.

When the client receives the `IBinder` in the `onServiceConnected()` callback, it must call *`YourServiceInterface`*`.Stub.asInterface(service)` to cast the returned parameter to *`YourServiceInterface`* type:

Kotlin Java

```
var iRemoteService: IRemoteService? = null
val mConnection = object : ServiceConnection {
    // Called when the connection with the service is established.
    override fun onServiceConnected(className: ComponentName, service: IBinder) {
        // Following the preceding example for an AIDL interface,
        // this gets an instance of the IRemoteInterface, which we can use to call
on the service.
        iRemoteService = IRemoteService.Stub.asInterface(service)
    }
    // Called when the connection with the service disconnects unexpectedly.
    override fun onServiceDisconnected(className: ComponentName) {
        Log.e(TAG, "Service has unexpectedly disconnected")
        iRemoteService = null
    }
}
```

For more sample code, see the `RemoteService.java` class in ApiDemos.

Passing objects over IPC

In Android 10 (API level 29 or higher), you can define `Parcelable` objects directly in AIDL. Types that are supported as AIDL interface arguments and other parcelables are also supported here. This avoids the additional work to manually write marshalling code and a custom class. However, this also creates a bare struct. If custom accessors or other functionality is desired, implement `Parcelable` instead.

```
package android.graphics;

// Declare Rect so AIDL can find it and knows that it implements
// the parcelable protocol.
parcelable Rect {
    int left;
    int top;
    int right;
    int bottom;
}
```

The preceding code sample automatically generates a Java class with integer fields `left`, `top`, `right`, and `bottom`. All relevant marshalling code is implemented automatically, and the object can be used directly without having to add any implementation.

You can also send a custom class from one process to another through an IPC interface. However, make sure the code for your class is available to the other side of the IPC channel and your class must support the `Parcelable` interface. Supporting `Parcelable` is important because it lets the Android system decompose objects into primitives that can be marshalled across processes.

To create a custom class that supports `Parcelable`, do the following:

1. Make your class implement the `Parcelable` interface.

2. Add a static field called `CREATOR` to your class that is an object implementing the `Parcelable.Creator` interface.
3. Finally, create an `.aidl` file that declares your parcelable class, as shown for the following `Rect.aidl` file.
   If you are using a custom build process, do *not* add the `.aidl` file to your build. Similar to a header file in the C language, this `.aidl` file isn't compiled.

AIDL uses these methods and fields in the code it generates to marshall and unmarshall your objects.

For example, here is a `Rect.aidl` file to create a `Rect` class that's parcelable:

```
package android.graphics;

// Declare Rect so AIDL can find it and knows that it implements
// the parcelable protocol.
parcelable Rect;
```

And here is an example of how the `Rect` class implements the `Parcelable` protocol.

The marshalling in the `Rect` class is straightforward. Take a look at the other methods on `Parcel` to see the other kinds of values you can write to a `Parcel`.

Warning: Remember the security implications of receiving data from other processes. In this case, the `Rect` reads four numbers from the `Parcel`, but it is up to you to ensure that these are within the acceptable range of values for whatever the caller is trying to do. For more information about how to keep your application secure from malware, see Security tips.

Methods with Bundle arguments containing Parcelables

If a method accepts a `Bundle` object that is expected to contain parcelables, make sure that you set the classloader of the `Bundle` by calling `Bundle.setClassLoader(ClassLoader)` before attempting to read from the `Bundle`. Otherwise, you run into `ClassNotFoundException` even though the parcelable is correctly defined in your application.
For example, consider the following sample `.aidl` file:

```
// IRectInsideBundle.aidl
package com.example.android;
/** Example service interface */
interface IRectInsideBundle {
    /** Rect parcelable is stored in the bundle with key "rect". */
    void saveRect(in Bundle bundle);
}
```

As shown in the following implementation, the `ClassLoader` is explicitly set in the `Bundle` before reading `Rect`:

Calling an IPC method

To call a remote interface defined with AIDL, take the following steps in your calling class:

Bear these points in mind when calling an IPC service:

- Objects are reference counted across processes.
- You can send anonymous objects as method arguments.

For more information about binding to a service, read the Bound services overview.

Here is some sample code that demonstrates calling an AIDL-created service, taken from the Remote Service sample in the ApiDemos project.

KotlinJava

```
private const val BUMP_MSG = 1
class Binding : Activity() {
    /** The primary interface you call on the service.  */
    private var mService: IRemoteService? = null
    /** Another interface you use on the service.  */
    internal var secondaryService: ISecondary? = null
    private lateinit var killButton: Button
    private lateinit var callbackText: TextView
    private lateinit var handler: InternalHandler
    private var isBound: Boolean = false
    /**
     * Class for interacting with the main interface of the service.
     */
    private val mConnection = object : ServiceConnection {
        override fun onServiceConnected(className: ComponentName, service:
IBinder) {
            // This is called when the connection with the service is
            // established, giving us the service object we can use to
            // interact with the service.  We are communicating with our
            // service through an IDL interface, so get a client-side
            // representation of that from the raw service object.
            mService = IRemoteService.Stub.asInterface(service)
            killButton.isEnabled = true
            callbackText.text = "Attached."
            // We want to monitor the service for as long as we are
            // connected to it.
            try {
                mService?.registerCallback(mCallback)
            } catch (e: RemoteException) {
                // In this case, the service crashes before we can
                // do anything with it. We can count on soon being
                // disconnected (and then reconnected if it can be restarted)
                // so there is no need to do anything here.
            }
            // As part of the sample, tell the user what happened.
            Toast.makeText(
                    this@Binding,
                    R.string.remote_service_connected,
                    Toast.LENGTH_SHORT
            ).show()
        }
        override fun onServiceDisconnected(className: ComponentName) {
            // This is called when the connection with the service is
            // unexpectedly disconnected&mdash;that is, its process crashed.
            mService = null
            killButton.isEnabled = false
            callbackText.text = "Disconnected."
            // As part of the sample, tell the user what happened.
            Toast.makeText(
                    this@Binding,
                    R.string.remote_service_disconnected,
                    Toast.LENGTH_SHORT
            ).show()
        }
    }
    /**
```

```
     * Class for interacting with the secondary interface of the service.
     */
    private val secondaryConnection = object : ServiceConnection {
        override fun onServiceConnected(className: ComponentName, service:
IBinder) {
            // Connecting to a secondary interface is the same as any
            // other interface.
            secondaryService = ISecondary.Stub.asInterface(service)
            killButton.isEnabled = true
        }
        override fun onServiceDisconnected(className: ComponentName) {
            secondaryService = null
            killButton.isEnabled = false
        }
    }
    private val mBindListener = View.OnClickListener {
        // Establish a couple connections with the service, binding
        // by interface names. This lets other applications be
        // installed that replace the remote service by implementing
        // the same interface.
        val intent = Intent(this@Binding, RemoteService::class.java)
        intent.action = IRemoteService::class.java.name
        bindService(intent, mConnection, Context.BIND_AUTO_CREATE)
        intent.action = ISecondary::class.java.name
        bindService(intent, secondaryConnection, Context.BIND_AUTO_CREATE)
        isBound = true
        callbackText.text = "Binding."
    }
    private val unbindListener = View.OnClickListener {
        if (isBound) {
            // If we have received the service, and hence registered with
            // it, then now is the time to unregister.
            try {
                mService?.unregisterCallback(mCallback)
            } catch (e: RemoteException) {
                // There is nothing special we need to do if the service
                // crashes.
            }
            // Detach our existing connection.
            unbindService(mConnection)
            unbindService(secondaryConnection)
            killButton.isEnabled = false
            isBound = false
            callbackText.text = "Unbinding."
        }
    }
    private val killListener = View.OnClickListener {
        // To kill the process hosting the service, we need to know its
        // PID.  Conveniently, the service has a call that returns
        // that information.
        try {
            secondaryService?.pid?.also { pid ->
                // Note that, though this API lets us request to
                // kill any process based on its PID, the kernel
                // still imposes standard restrictions on which PIDs you
                // can actually kill. Typically this means only
```

```
                // the process running your application and any additional
                // processes created by that app, as shown here. Packages
                // sharing a common UID are also able to kill each
                // other's processes.
                Process.killProcess(pid)
                callbackText.text = "Killed service process."
            }
        } catch (ex: RemoteException) {
            // Recover gracefully from the process hosting the
            // server dying.
            // For purposes of this sample, put up a notification.
            Toast.makeText(this@Binding, R.string.remote_call_failed,
Toast.LENGTH_SHORT).show()
        }
    }
    // ---------------------------------------------------------------------
    // Code showing how to deal with callbacks.
    // ---------------------------------------------------------------------
    /**
     * This implementation is used to receive callbacks from the remote
     * service.
     */
    private val mCallback = object : IRemoteServiceCallback.Stub() {
        /**
         * This is called by the remote service regularly to tell us about
         * new values.  Note that IPC calls are dispatched through a thread
         * pool running in each process, so the code executing here is
         * NOT running in our main thread like most other things. So,
         * to update the UI, we need to use a Handler to hop over there.
         */
        override fun valueChanged(value: Int) {
            handler.sendMessage(handler.obtainMessage(BUMP_MSG, value, 0))
        }
    }
    /**
     * Standard initialization of this activity.  Set up the UI, then wait
     * for the user to interact with it before doing anything.
     */
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
setContentView(R.layout.remote_service_binding)
        // Watch for button taps.
        var button: Button = findViewById(R.id.bind)
        button.setOnClickListener(mBindListener)
        button = findViewById(R.id.unbind)
        button.setOnClickListener(unbindListener)
        killButton = findViewById(R.id.kill)
        killButton.setOnClickListener(killListener)
        killButton.isEnabled = false        callbackText =
findViewById(R.id.callback)
        callbackText.text = "Not attached."
        handler = InternalHandler(callbackText)
    }
    private class InternalHandler(
            textView: TextView,
            private val weakTextView: WeakReference<TextView> =
```

```
WeakReference(textView)
    ) : Handler() {
        override fun handleMessage(msg: Message) {
            when (msg.what) {
                BUMP_MSG -> weakTextView.get()?.text = "Received from service:
${msg.arg1}"
                else -> super.handleMessage(msg)
            }
        }
    }
}
```