Android Services & Local IPC: Advanced Bound Service Communication – AIDL Syntax & Supported Data Types

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Module

• Understand the Android interface syntax & supported data types

```java
interface IDownload {
    oneway void setCallback(in IDownloadCallback callback);
}

interface IDownloadCallback {
    oneway void sendPath(in String path);
}
```
AIDL Syntax & Supported Data Types

- AIDL allows application developers to declare their “business” logic methods using a Java-like interface syntax.

```java
interface IDropBoxManagerService {
    void add(in DropBoxManager.Entry entry);
    ...
    DropBoxManager.Entry getNextEntry(String tag, long millis);
}
```

frameworks/base/core/java/com/android/internal/os/IDropBoxManagerService.aidl
AIDL Syntax & Supported Data Types

- AIDL allows application developers to declare their “business” logic methods using a Java-like interface syntax

```java
interface IDropBoxManagerService {
    void add(in DropBoxManager.Entry entry);
    ...
    DropBoxManager.Entry getNextEntry(String tag, long millis);
}
```

- Similarities with Java interfaces
  - AIDL can declare methods with typed parameters & a return value
AIDL Syntax & Supported Data Types

- AIDL allows application developers to declare their “business” logic methods using a Java-like interface syntax

```java
interface IDropBoxManagerService {
    void add(in DropBoxManager.Entry entry);
    ...
    DropBoxManager.Entry getNextEntry(String tag, long millis);
}
```

- Similarities with Java interfaces
- Differences from Java interfaces
  - No static fields
AIDL Syntax & Supported Data Types

- AIDL allows application developers to declare their "business" logic methods using a Java-like interface syntax

```java
interface IDropBoxManagerService {
    void add(in DropBoxManager.Entry entry);
    ...
    DropBoxManager.Entry getNextEntry(String tag, long millis);
}
```

- Similarities with Java interfaces
- Differences from Java interfaces
  - No static fields
  - All non-primitive parameters must be labeled by "direction"
    - `in` – (default/only mode for primitives) transferred to remote method
    - `out` – returned to the caller
    - `inout` – both in & out (rarely used)

Limit direction to just what’s needed since marshaling parameters is expensive
AI DL Syntax & Supported Data Types

- AI DL allows application developers to declare their “business” logic methods using a Java-like interface syntax

```java
interface IDropBoxManagerService {
    void add(in DropBoxManager.Entry entry);
    ...
    DropBoxManager.Entry getNextEntry(String tag, long millis);
}
```

- Similarities with Java interfaces
- Differences from Java interfaces
  - No static fields
  - All non-primitive parameters must be labeled by “direction”
- Methods (& AI DL interfaces themselves) can be defined as `oneway`
  - `oneway` method invocations don’t block the caller
AIDL Syntax & Supported Data Types

- AIDL allows application developers to declare their “business” logic methods using a Java-like interface syntax

```java
interface IDropBoxManagerService {
    void add(in DropBoxManager.Entry entry);
    ...
    DropBoxManager.Entry getNextEntry(String tag, long millis);
}
```

- Similarities with Java interfaces
- Differences from Java interfaces
  - No static fields
  - All non-primitive parameters must be labeled by “direction”
  - Methods (& AIDL interfaces themselves) can be defined as oneway
  - Methods cannot throw exceptions
AIDL Syntax & Supported Data Types

• AIDL allows application developers to declare their “business” logic methods using a Java-like interface syntax

```java
interface IDropBoxManagerService {
    void add(in DropBoxManager.Entry entry);
    ... 
    DropBoxManager.Entry getNextEntry(String tag, long millis);
}
```

• Similarities with Java interfaces

• Differences from Java interfaces
  • No static fields
  • All non-primitive parameters must be labeled by “direction”
  • Methods (& AIDL interfaces themselves) can be defined as oneway
  • Methods cannot throw exceptions
  • Interfaces can’t inherit from other interfaces

[developer.android.com/guide/components/aidl.html](https://developer.android.com/guide/components/aidl.html) has more info
AIDL Syntax & Supported Data Types

- AIDL allows application developers to declare their “business” logic methods using a Java-like interface syntax

- Supported Java primitives
  - boolean, boolean[], byte, byte[], char[], int, int[], long, long[], float, float[], double, double[]
  - java.lang.CharSequence, java.lang.String

```java
interface IEmailService {
    ...
    boolean createFolder(long accountId, String name);
    boolean deleteFolder(long accountId, String name);
    boolean renameFolder(long accountId, String oldName, String newName);
}
```

`packages/apps/Email/emailcommon/src/com/android/emailcommon/service/IEmailService.aidl`
AIDL Syntax & Supported Data Types

- AIDL allows application developers to declare their “business” logic methods using a Java-like interface syntax
- Supported Java primitives
  - java.util.List
    - Uses java.util.ArrayList internally
    - List elements must be valid
  - Generic lists supported

```java
oneway interface INetworkQueryServiceCallback {
    void onQueryComplete
        (in List<OperatorInfo> networkInfoArray,
         int status);
}
```

`packages/apps/Phone/src/com/android/phone/INetworkQueryServiceCallback.aidl`
AIDL Syntax & Supported Data Types

- AIDL allows application developers to declare their “business” logic methods using a Java-like interface syntax.
- Supported Java primitives:
  - java.util.List
  - java.util.Map
    - Uses java.util.HashMap internally
    - Map elements must be valid
  - Java.util.Map
    - Generic maps *not* supported
    - Not widely used (no use in Android)
AI DL Syntax & Supported Data Types

- AIDL allows application developers to declare their “business” logic methods using a Java-like interface syntax
- Supported Java primitives
  - `java.util.List`
  - `java.util.Map`
- Classes implementing the `Parcelable` protocol

```java
public class StatusBarIcon implements Parcelable {
    ...  
    public void readFromParcel(Parcel in) {
        ... }
    public void writeToParcel(Parcel out, int flags) {
        ... }
}
```

`Parcelable StatusBarIcon;`

```java
oneway interface IStatusBar {
    void setIcon(int index, in StatusBarIcon icon);
    ... 
}
```

[developer.android.com/guide/components/aidl.html#PassingObjects](https://developer.android.com/guide/components/aidl.html#PassingObjects) has more
AIDL Syntax & Supported Data Types

- AIDL allows application developers to declare their “business” logic methods using a Java-like interface syntax.
- Supported Java primitives:
  - java.util.List
  - java.util.Map
- Classes implementing the Parcelable protocol:

```
public class StatusBarIcon
    implements Parcelable {
    ...
    public void readFromParcel(Parcel in) {
        ... }
    public void writeToParcel(Parcel out, int flags) {
        ... }
}
```

```
parcelable StatusBarIcon;
```

```
oneway interface IStatusBar {
    void setIcon(int index, in StatusBarIcon icon);
    ...
}
```

```
AIDL source file
```
```
Java source file
```

```
frameworks/base/core/java/com/android/internal/statusbar/IStatusBar.aidl
```

Douglas C. Schmidt
Summary

- AIDL uses a simple syntax that lets you declare an interface with one or more methods that can take parameters & return values

```
InterfaceDeclaration:
    interface Identifier InterfaceBody
InterfaceBody:
    { { InterfaceBodyDeclaration } }  
InterfaceBodyDeclaration:
    ;
    InterfaceMethodDecl
InterfaceMethodDecl:
    Type Identifier InterfaceMethodDeclaratorRest
InterfaceMethodDeclaratorRest:
    FormalParameters
```

AIDL interfaces are a subset of Java interfaces
Summary

- AIDL uses a simple syntax that lets you declare an interface with one or more methods that can take parameters & return values
- The parameters & return values can be of any supported type, even other AIDL-generated interfaces
- Interface methods that are passed parameters of other interfaces are commonly used to implement asynchronous one-way callbacks
Summary

- AIDL uses a simple syntax that lets you declare an interface with one or more methods that can take parameters & return values.
- The parameters & return values can be of any supported type, even other AIDL-generated interfaces.
- You must construct the .aidl file using a subset of the Java programming language.
- Each .aidl file must define a single interface & requires only the interface declaration & method signatures.

```java
interface IDownload {
    oneway void setCallback(in IDownloadCallback callback);
}
```

```java
interface IDownloadCallback {
    oneway void sendPath(in String path);
}
```
Android Services & Local IPC: 
Advanced Bound Service Communication 
– Implementing AIDL Interfaces

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt
Learning Objectives in this Part of the Module

- Understand how to implement AIDL interfaces via Eclipse

Diagram:

Client Process:
- DownloadActivity
- initiateDownload()
- downloadImage()
- libbinder
- Dalvik Runtime
- Native Runtime

Server Process:
- DownloadService
- libbinder
- Dalvik Runtime
- Native Runtime

Steps:
1. Client calls downloadImage() on the Stub.Proxy
2. Proxy performs parameter marshaling & calls mRemote.transact()
3. Submit transaction data to Linux kernel via blocking ioctl() call
4. Wakeup from blocking ioctl() call & get transaction data
5. Get target Binder object & call onTransact() on its Stub
6. Stub performs parameter demarshaling & calls downloadImage()
Developing with AIDL on Eclipse

- Each Binder-based service is defined in a separate .aidl file & saved in a src directory
- Eclipse ADT automatically calls aidl for each .aidl file it finds in a src directory
Developing with AIDL on Eclipse

- Each Binder-based service is defined in a separate .aidl file & saved in a src directory.
- The Android aidl build tool extracts a real Java interface from each .aidl file & places it into a *.java file in the gen directory.
- This *.java file also contains:
  - A generated Stub that extends Android’s android.os.IBinder
  - A Proxy that inherits from the AIDL interface.

developer.android.com/guide/components/aidl.html has more info.
Structure of AIDL-based Solutions

```
interface IDownload {
    String downloadImage (String uri);
}
```

**Generated by AIDL compiler**

- **Defined by the App**
  - Client-related

- **Server-related**
  - **IBinder interface**
  - **Binder class**
  - **Stub inner class**
  - **Stub.Proxy inner class**

**AIDL Compiler**

- **Defined by Android**
  - Class that implements the Idownload interface
Example of Generated Stub

```
public interface IDownload extends android.os.IInterface {
    public static abstract class Stub extends android.os.Binder
        implements IDownload {

            Local-side IPC
            implementation class

            public Stub() {
                this.attachInterface(this, DESCRIPTOR);
            }

            ... Construct the stub & attach it to the interface
```
Example of Generated Stub

public interface IDownload extends android.os.IInterface {
    public static abstract class Stub extends android.os.Binder
        implements IDownload {

        public static IDownload asInterface(android.os.IBinder obj) {
            if ((obj==null)) return null;
            android.os.IInterface iin = (android.os.IInterface)
                obj.queryLocalInterface(DESCRIPTOR);
            if (((iin != null) && (iin instanceof IDownload)))
                return ((IDownload)iin);
            return new IDownload.Stub.Proxy(obj);
        }

        ...

        Cast an IBinder object into an IDownload interface, generating a proxy if needed
Example of Generated Proxy

```java
public interface IDownload extends android.os.IInterface {
    public static abstract class Stub ... {
        private static class Proxy implements IDownload {

            private android.os.IBinder mRemote;

            Proxy(android.os.IBinder remote) {
                mRemote = remote;
            }

            ... Cache Binder for subsequent use by Proxy
        }
    }
}
```

*Used by a client to call a remote method*
Example of Generated Proxy

```
public interface IDownload extends android.os.IInterface {
    public static abstract class Stub ... {
        private static class Proxy implements IDownload {
            ...

            public String downloadImage(String uri) ... {
                android.os.Parcel _data = android.os.Parcel.obtain();
                android.os.Parcel _reply = android.os.Parcel.obtain();
                java.lang.String _result;
                _data.writeInterfaceToken(DESCRIPTOR);
                _data.writeString(url);
                mRemote.transact(Stub.TRANSACTION_downloadImage, _data,
                                _reply, 0);
                _reply.readException();
                _result = _reply.readString();
                ...
                return _result;
            }
        }
    }
}
```

This code fragment has been simplified a bit to fit onto the slide.

Marshal the parameter, transmit to the remote object, & demarshal the result.
public interface IDownload extends android.os.IInterface {
    public static abstract class Stub extends android.os.Binder
    implements IDownload {

        This method is dispatched by Binder RPC to trigger a callback on our downloadImage()

    public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int flags) ... {
        switch (code) {
            case TRANSACTION_downloadImage:
                data.enforceInterface(DESCRIPTOR);
                java.lang.String _arg0 = data.readString();
                java.lang.String _result = this.downloadImage(_arg0);
                reply.writeNoException();
                reply.writeString(_result);
                return true;
            ...  

        Demarshal the parameter, dispatch the upcall, & marshal the result

        This code fragment has been simplified a bit to fit onto the slide

    }
Implementing an AIDL Interface

• Given an auto-generated AIDL stub, you must implement certain methods

```java
public interface IDownload extends android.os.IInterface {
    public static abstract class Stub extends android.os.Binder implements IDownload {
        ... 
        public String downloadImage(String uri)
            throws android.os.RemoteException;
    }
}
```

• Either implement downloadImage() directly in the stub or by forwarding the stub to some other implementation method
Implementing an AIDL Interface

- Given an auto-generated AIDL stub, you must implement certain methods
- Implementation steps:
  1. Create a private instance of AIDL-generated Stub class

```java
public class DownloadService
    extends Service {
    private final IDownload.Stub
        binder = null;
    public void onCreate() {
        binder = new IDownload.Stub(){
            public String downloadImage
                (String uri) {
                ...
            }
        };

        public IBinder onBind
            (Intent intent)
        { return this.binder; }
    }
```
Implementing an AIDL Interface

- Given an auto-generated AIDL stub, you must implement certain methods.

Implementation steps:

1. Create a private instance of AIDL-generated Stub class

```
public class DownloadService
    extends Service {
    private final IDownload.Stub
        binder = null;
    public void onCreate() {
        binder = new IDownload.Stub() {
            public String downloadImage
                (String uri) {
                ...
            }
        };

        public IBinder onBind
            (Intent intent) {
            return this.binder;
        }
    }
```

2. Implement Java methods for each method in the AIDL file...
Implementing an AIDL Interface

- Given an auto-generated AIDL stub, you must implement certain methods
- Implementation steps:
  1. Create a private instance of AIDL-generated Stub class
  2. Implement Java methods for each method in the AIDL file
  3. Return this private instance from your onBind() method in the Service subclass

```java
public class DownloadService extends Service {
    private final IDownload.Stub binder = null;
    public void onCreate() {
        binder = new IDownload.Stub() {
            public String downloadImage(String uri) {
                ...
            }
        };
    }

    public IBinder onBind(Intent intent) {
        return this.binder;
    }
}
```
Putting the Pieces Together End-to-End

Client Process

DownloadActivity

initiateDownload()

downloadImage()

IDownload. Stub.Proxy

libbinder
Dalvik Runtime
Native Runtime

Server Process

DownloadService

downloadImage()

IDownload. Stub

libbinder
Dalvik Runtime
Native Runtime

Client calls downloadImage() on the Stub.Proxy
Putting the Pieces Together End-to-End

Client Process

**DownloadActivity**
- `initiateDownload()`
- `downloadImage()`

**libbinder**
- Dalvik Runtime
- Native Runtime

Server Process

**DownloadService**
- `downloadImage()`
- `IDownload.Stub.Proxy`
- `transact()`

**libbinder**
- Dalvik Runtime
- Native Runtime

proxy performs parameter marshaling & calls mRemote.

---

**Linux Kernel**
- Display Driver
- Camera Driver
- Bluetooth Driver
- Shared Memory Driver
- Binder (IPC) Driver
- USB Driver
- Keypad Driver
- WiFi Driver
- Audio Drivers
- Power Management
Putting the Pieces Together End-to-End

Client Process

DownloadActivity

initiateDownload()

downloadImage()

IDownload. Stub.Proxy

libbinder

Dalvik Runtime

Native Runtime

Server Process

DownloadService

downloadImage()

IDownload. Stub

libbinder

Dalvik Runtime

Native Runtime

Submit transaction data to Linux kernel via blocking ioctl() call
Putting the Pieces Together End-to-End

Client Process
- DownloadActivity
  - initiateDownload()
  - downloadImage()
- libbinder
- Dalvik Runtime
- Native Runtime

Server Process
- DownloadService
  - downloadImage()
- libbinder
- Dalvik Runtime
- Native Runtime

Wakeup from blocking ioctl() call & get transaction data
Putting the Pieces Together End-to-End

Client Process

DownloadActivity

initiateDownload()

downloadImage()

IDownload.Stub.Proxy

libbinder

Dalvik Runtime

Native Runtime

Server Process

DownloadService

downloadImage()

Get target Binder object & call onTransact() on its Stub

libbinder

Dalvik Runtime

Native Runtime
Putting the Pieces Together End-to-End

**Client Process**

- DownloadActivity
  - initiateDownload()
  - downloadImage()
  - IDownload.Stub.Proxy
  - libbinder
  - Dalvik Runtime
  - Native Runtime

**Server Process**

- DownloadService
  - downloadImage()
  - IDownload.Stub
  - libbinder
  - Dalvik Runtime
  - Native Runtime

**Stub performs parameter demarshaling & calls downloadImage()**
Putting the Pieces Together End-to-End

Client Process

DownloadActivity

initiateDownload()

downloadImage()

IDownload.Stub.Proxy

libbinder

Dalvik Runtime

Native Runtime

Server Process

DownloadService

downloadImage()

IDownload.Stub

libbinder

Dalvik Runtime

Native Runtime

Stub marshals the result from downloadImage() into a reply parcel
Putting the Pieces Together End-to-End

The Stub returns the reply parcel back to the Linux kernel.
Putting the Pieces Together End-to-End

The reply parcel is passed through the Linux kernel back to the client.
Putting the Pieces Together End-to-End

The reply parcel arrives back at the Proxy, which returns the result back to the caller of `downloadImage()`.
**AI DL & Binder RPC Call Semantics**

- Calls made from a local process are executed in the same thread that makes the call.
- If this is the main UI thread, that thread continues to execute in the AIDL interface.
Calls made from a local process are executed in the same thread that makes the call:

- If this is the main UI thread, that thread continues to execute in the AIDL interface.
- If it is another thread, that is the one that executes the code in the Service.
Calls made from a local process are executed in the same thread that makes the call.

Calls from a remote process are dispatched from a thread pool the platform maintains inside of a process.

An implementation of an AIDL interface must therefore be completely thread-safe.
AI DL & Binder RPC Call Semantics

- Calls made from a local process are executed in the same thread that makes the call.
- Calls from a remote process are dispatched from a thread pool the platform maintains inside of a process.
- The `oneway` keyword modifies the behavior of remote calls.
  - When used, a remote call does not block—it simply sends the transaction data & returns immediately.
Android Services & Local IPC

Al DL & Binder RPC Call Semantics

- Calls made from a local process are executed in the same thread that makes the call
- Calls from a remote process are dispatched from a thread pool the platform maintains inside of a process
- The `oneway` keyword modifies the behavior of remote calls
  - When used, a remote call does not block—it simply sends the transaction data & returns immediately
  - If `oneway` is used with a local call, the call is still synchronous
    - But no results are returned

developer.android.com/guide/components/aidl.html
Summary

- AIDL is an interface definition language used to generate code that enables two processes on an Android device to interact using IPC
- If code in a process calls methods on an object in another process, AIDL can generate code to (de)marshal parameters passed between processes
Summary

- AIDL is an interface definition language used to generate code that enables two processes on an Android device to interact using IPC.
- AIDL is interface-based, similar to CORBA & Java, but lighter weight.
  - It uses a proxy to pass values between a client & Bound Service.
Summary

- AIDL is an interface definition language used to generate code that enables two processes on an Android device to interact using IPC.
- AIDL is interface-based, similar to CORBA, but lighter weight.
- There are hundreds of *.aidl files used in Android.

AIDL is often used to mediate two-way communication between Activity & Service or Service & Service.
Android Services & Local IPC: Advanced Bound Service Communication – Programming Apps with AIDL

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Module

- Understand how to program an App using AIDL & Bound Services
- Both synchronous

### Diagram

**Process A**
- **ServiceConnection**
  - `onStart()`: bindService()
  - `onClick()`: mGenerator

**Process B**
- **KeyGenService**
  - `onBind()`: mBinder
  - `getKey()`: dispatch callback
  - `onStart()`: bindService()

**Binder IPC Mechanism**
- `Intent`
  - Invoke `getKeay()`

**Steps**
1. Start Bound Service process if it's not already running
2. Dispatch callback
3. Return ref to Binder object
4. onServiceConnected()
5. Assign Proxy to data member
6. Invoke `getKey()`
7. Stub dispatches method & returns result
Learning Objectives in this Part of the Module

- Understand how to program an App using AIDL & Bound Services
- Both synchronous & asynchronous

Process A

KeyUserAsyncActivity
- onClick()
- onStart()
- bindService()

ServiceConnection
- onServiceConnected()
- KeyGeneratorCallback
- sendKey()

Process B

KeyGenAsyncService
- onBind()

Binder IPC Mechanism
- Intent

Invoke setCallback()

1. Start Bound Service process if it's not already running
2. Dispatch callback
3. Return ref to Binder object
4. Dispatch callback
5. Assign Proxy to data member
6. Invoke setCallback()
7. Stub dispatches method
8. Return result
ID Service via Synchronous AIDL

- Client uses a “Bound Service” hosted in another App
- Requires IPC via AIDL & the Android Binder RPC mechanism
ID Service via Synchronous AIDL

- Client uses a “Bound Service” hosted in another App
- Client needs an ID from service
ID Service via Synchronous AIDL

- Client uses a “Bound Service” hosted in another App
- Client needs an ID from service
- Overall structure of this solution is similar to the Messenger solution presented earlier
  - Main difference is that the AIDL calls are synchronous, statically typed, & thread-safe, whereas the Messenger calls are asynchronous, dynamically typed, & sequential
ID Service via Synchronous AIDL

- Client uses a “Bound Service” hosted in another App
- Client needs an ID from service
- Overall structure of this solution is similar to the Messenger solution presented earlier
  - Main difference is that the AIDL calls are synchronous, statically typed, & thread-safe, whereas the Messenger calls are asynchronous, dynamically typed, & sequential
- We’ll also show an asynchronous AIDL example shortly
Define a Remote Interface

• Declare interface in an .aidl file

```java
package course.examples.
  Services.KeyCommon;

interface KeyGenerator {
  String getKey();
}
```
Using a Synchronous AIDL in a Bound Service

- Enables synchronous interaction between an Activity & a Bound Service using AIDL & Binder RPC
Using a Synchronous AIDL in a Bound Service

- Enables synchronous interaction between an Activity & a Bound Service using AIDL & Binder RPC

![Diagram showing interaction between KeyUserActivity and KeyGenService]
Using a Synchronous AIDL in a Bound Service

- Enables synchronous interaction between an Activity & a Bound Service using AIDL & Binder RPC
Using a Synchronous AIDL in a Bound Service

- Enables synchronous interaction between an Activity & a Bound Service using AIDL & Binder RPC
Compile .aidl File to Generate Stub & Proxy

- Generate a Java interface with same name as .aidl file
- Eclipse does this automatically

```java
interface KeyGenerator {
    String getKey();
}
```

Client Process

KeyUserActivity

onClick()

Binder Framework

Dalvik Runtime

Native Runtime

Server Process

KeyGenService

IDL Compiler

Binder Framework

Dalvik Runtime

Native Runtime
Compile .aidl File to Generate Stub & Proxy

- Generate a Java interface with same name as .aidl file
- Generated interface contains:
  - Abstract inner class called Stub & a Proxy class
  - Interface & helper methods

```java
interface KeyGenerator {
    String getKey();
}
```
Implement Remote Methods

```java
public class KeyGenService extends Service {
    private Set<UUID> keys = new HashSet<UUID>();

    private final KeyGenerator.Stub mBinder =
        new KeyGenerator.Stub() {
            public String getKey() {
                UUID id;
                synchronized (keys) {
                    do { id = UUID.randomUUID(); } while (keys.contains(id));
                    keys.add(id);
                }
                return id.toString();
            }
        };

    public IBinder onBind(Intent intent) { return this.mBinder; }
}
```

- Unique set of keys
- Generate unique ID in a thread-safe manner & return it
- Return reference to the Binder object
Implement the ID Client Activity

```java
public class KeyUserActivity extends Activity {
    private KeyGenerator mGenerator;
    private boolean mBound;

    private ServiceConnection connection =
            new ServiceConnection() {
                public void onServiceConnected(ComponentName className, IBinder iBinder) {
                    mGenerator = KeyGenerator.Stub.asInterface(iBinder);
                    mBound = true;
                }
                ...

                public void onServiceDisconnected(ComponentName className) {
                    mGenerator = null; mBound = false;
                }
            }

    // Remote Service callback methods

    // Proxy to the remote Bound Service

    // Local state
```
Implement the ID Client Activity

```java
public class KeyUserActivity extends Activity {
  ...
  protected void onStart() {
    super.onStart();

    Intent intent = new Intent(KeyGenService.class.getName());
    bindService(intent, this.connection,
                Context.BIND_AUTO_CREATE);
  }

  protected void onStop() {
    if (mBound) unbindService(this.connection);
    super.onStop();
  }
}
```

*Use name of KeyGenService as an ACTION*

*Bind to Service*

*Unbind from Service*
public class KeyUserActivity extends Activity {
    TextView output; ...
    public void onCreate(Bundle icicle) {
        ...
        output = (TextView)findViewById(R.id.output);
        final Button goButton = (Button)findViewByld(R.id.go_button);

        goButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                try {
                    if (bound)
                        output.setText(mGenerator.getKey());
                } catch (RemoteException e) {
                }
            }
        });
    }
}
Service AndroidManifest.xml

• The Service AndroidManifest.xml file must be registered with Android before the client Activity tries to run

```xml
<manifest ...
    package="course.examples.Services.KeyGenService">
    <application "...">
        <service android:name=".KeyGenService"
            android:exported="true">
            <intent-filter>
                <action android:name="course.examples.Services.KeyGenService"/>
            </intent-filter>
        </service>
    </application>
</manifest>
```

Use name of the Service as a filter!
• The client Activity AndroidManifest.xml file has the typical format

```xml
<manifest ... package="course.examples.Services.KeyClient">
  <application "">
    <activity android:name=".KeyUser" />
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```
ID Service via Asynchronous AIDL

- Client uses a “Bound Service” hosted in another App
- Requires IPC via AIDL & the Android Binder RPC mechanism
ID Service via Asynchronous AIDL

- Client uses a “Bound Service” hosted in another App
- Client needs an ID from service
ID Service via Asynchronous AIDL

- Client uses a “Bound Service” hosted in another App
- Client needs an ID from service
- Overall structure of this solution is similar to the AIDL solution presented earlier
  - Main difference is that the AIDL calls in this example are one-way asynchronous
  - In contrast, the previous AIDL example calls are two-way synchronous
ID Service via Asynchronous AIDL

- Client uses a “Bound Service” hosted in another App
- Client needs an ID from service
- Overall structure of this solution is similar to the AIDL solution presented earlier
  - Main difference is that the AIDL calls in this example are one-way asynchronous
- This solution is also similar to the asynchronous Messenger example
  - The main difference is that the calls are statically typed, more “stylized,” & multi-threaded
Define Two Remote Interfaces

- Declare interface in two .aidl files since you can’t define more than one interface in each *.aidl file

**KeyGeneratorCallback.aidl**

```java
package course.examples.Services.KeyCommon;

interface KeyGeneratorCallback {
    oneway void sendKey(in String key);
}
```

**KeyGenerator.aidl**

```java
package course.examples.Services.KeyCommon;

import course.examples.Services.KeyCommon.KeyGeneratorCallback;

interface KeyGenerator {
    oneway void setCallback(in KeyGeneratorCallback callback);
}
```
Define Two Remote Interfaces

- Declare interface in two .aidl files since you can’t define more than one interface in each *.aidl file

**KeyGeneratorCallback.aidl**
```java
package course.examples.Services.KeyCommon;

interface KeyGeneratorCallback {
    oneway void sendKey(in String key);
}
```

**KeyGenerator.aidl**
```java
package course.examples.Services.KeyCommon;

import course.examples.Services.KeyCommon.KeyGeneratorCallback;

interface KeyGenerator {
    oneway void setCallback(in KeyGeneratorCallback callback);
}
```

- When `oneway` is used on a remote call it does not block
- It simply sends the transaction data & immediately returns
Using an Asynchronous AIDL in a Bound Service

- Enables asynchronous interaction between an Activity & a Bound Service using AIDL & Binder RPC
Enables asynchronous interaction between an Activity & a Bound Service using AIDL & Binder RPC.
Using an Asynchronous AIDL in a Bound Service

- Enables asynchronous interaction between an Activity & a Bound Service using AIDL & Binder RPC

![Diagram showing the interaction between KeyUserAsyncActivity and KeyGenAsyncService.](image)

1. KeyUserAsyncActivity invokes setCallback().
2. KeyGenAsyncService binds ServiceConnection.
3. KeyGenAsyncService invokes onBind() & sets mBinder.
4. KeyUserAsyncActivity invokes onClick().
5. KeyUserAsyncActivity invokes onStart().
6. ServiceConnection invokes onServiceConnected().
7. mGenerator invokes sendKey().
8. Send result via callback.
Implement Remote Methods

```java
public class KeyGenAsyncService extends Service {
    private Set<UUID> keys = new HashSet<UUID>();

    public void setCallback(final keyGeneratorCallback callback) {
        UUID id;
        synchronized (keys) {
            do { id = UUID.randomUUID(); } while (keys.contains(id));
            keys.add(id);
        }
        final String key = id.toString();
        ...
        callback.sendKey(key);
        ...
    }

    public IBinder onBind(Intent intent) { return this.mBinder; }
}
```

- Unique set of keys
- Runs in a thread from Binder’s thread pool
- Generate unique ID
- Make a one-way (non-blocking) call to return result
- Return reference to the Binder object

Since setCallback() is a one-way it doesn’t block the Activity caller
Implement the Asynchronous ID Client Activity

code:

```java
public class KeyUserAsyncActivity extends Activity {
    private KeyGenerator mGenerator;
    private boolean mBound;

    private ServiceConnection connection =
            new ServiceConnection() {
                public void onServiceConnected(ComponentName className,
                        IBinder iBinder) {
                    mGenerator = KeyGenerator.Stub.asInterface(iBinder);
                    mBound = true;
                }
                ...

                public void onServiceDisconnected(ComponentName className){
                    mGenerator = null; mBound = false;
                }
            };
```
Implement the Asynchronous ID Client Activity

```java
public class KeyUserAsyncActivity extends Activity {
    ...

    protected void onStart() {
        super.onStart();

        Intent intent = new Intent(KeyGenService.class.getName());
        bindService(intent, this.connection, Context.BIND_AUTO_CREATE);
    }

    protected void onStop() {
        if (mBound) unbindService(this.connection);
        super.onStop();
    }
}
```

Use name of `KeyGenService` as an ACTION

Bind to Service

Unbind from Service
public class KeyUserAsyncActivity extends Activity {
    TextView output; ... 
    public void onCreate(Bundle icicle) {
        ... 
        output = (TextView)findViewById(R.id.output);
        final Button goButton = (Button)findViewByld(R.id.go_button);
        goButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                try {
                    if (bound)
                        mGenerator.setCallback(mCallback);
                } catch (RemoteException e) {}
            }
        });
    }
}
public class KeyUserAsyncActivity extends Activity {
    ...
    private final KeyGeneratorCallback.Stub mCallback =
            new KeyGeneratorCallback.Stub() {

    
        public void sendKey (final String key) {
            runOnUiThread (new Runnable() {
                public void run() {
                    output.setText(key);
                }
            });
        }
    }

    Since sendKey() is a one-way it doesn’t block the Service caller
Summary

- AIDL implements **Broker**, Messenger implements **Command Processor**
Summary

- AIDL implements *Broker*, Messenger implements *Command Processor*
- When you need to perform complex IPC interactions, using AIDL for your interface is often easier than implementing Messengers
- AIDL calls are two-way synchronous, Services processes calls concurrently
AIDL implements Broker, Messenger implements Command Processor.

When you need to perform complex IPC interactions, using AIDL for your interface is often easier than implementing Messengers.

AIDL calls are two-way synchronous, Services processes calls concurrently.

AIDL also supports asynchrony via callback objects.
Summary

- AIDL implements *Broker*, Messenger implements *Command Processor*
- When you need to perform complex IPC interactions, using AIDL for your interface is often easier than implementing Messengers
- When you need to perform simple IPC interactions, using a Messenger for your interface is often easier than implementing it with AIDL
- A Messenger queues all calls, which Service then handles asynchronously
Summary

- AIDL implements *Broker*, Messenger implements *Command Processor*
- When you need to perform complex IPC interactions, using AIDL for your interface is often easier than implementing Messengers
- When you need to perform simple IPC interactions, using a Messenger for your interface is often easier than implementing it with AIDL
- For many Apps a Service doesn't need to perform multi-threading
  - Using a Messenger allows the service to handle one call at a time without extra programming effort
Summary

- AIDL implements *Broker*, Messenger implements *Command Processor*
- When you need to perform complex IPC interactions, using AIDL for your interface is often easier than implementing Messengers
- When you need to perform simple IPC interactions, using a Messenger for your interface is often easier than implementing it with AIDL
- For many Apps a Service doesn't need to perform multi-threading
  - Using a Messenger allows the service to handle one call at a time without extra programming effort
  - If it's important that your service be multi-threaded, then you should use AIDL to define your interface

```java
interface KeyGenerator {
    String getKey();
}
```