./

Learning Report – Android App Components – Services, Local IPC and Content Providers MOOC



|  |  |  |
| --- | --- | --- |
| **Name** | **PS Number** | **Email ID** |
| Praveen Kumar Chaubey | 99002503 | Praveen.chaubey@ltts.com |

Table of Contents

[Table of Contents 3](#_Toc54712553)

[Table of Figures 3](#_Toc54712554)

[Layered architecture 5](#_Toc54712555)

[Android Layered Architecture Overview 6](#_Toc54712556)

[Linux Kernel 7](#_Toc54712557)

[Memory management: 7](#_Toc54712558)

[Process Management: 8](#_Toc54712559)

[Kernel Extension: 9](#_Toc54712560)

[Android Hardware Abstraction Layer: 10](#_Toc54712561)

[Local and Remote IPC 10](#_Toc54712562)

[Android Libraries: 12](#_Toc54712563)

[Android Core and Native Libraries 12](#_Toc54712564)

[Object oriented Framework: 13](#_Toc54712565)

[Android Services 14](#_Toc54712566)

[Model View Presenter 14](#_Toc54712567)

[Services 16](#_Toc54712568)

[Started Services 17](#_Toc54712569)

[Bound Services 18](#_Toc54712570)

[Hybrid of Started and Bound Service 19](#_Toc54712571)

[Android Service Implementation 20](#_Toc54712572)

[Service Lifecycle States 23](#_Toc54712573)

[Service Lifecycle Operations 23](#_Toc54712574)

[Starting an Android Service 24](#_Toc54712575)

[Methods used to start Android Service 25](#_Toc54712576)

[Integrating a service into an App 25](#_Toc54712577)

[Concurrent Programming with Services 26](#_Toc54712578)

[Intent Service Framework structure and functionality 27](#_Toc54712579)

[IntentService Usage Considerations 28](#_Toc54712580)

[Android Service Deployment Models 29](#_Toc54712581)

[Android Service Communication Models 30](#_Toc54712582)

[References: 31](#_Toc54712583)

# Table of Figures

Figure 1: Layered architecture 6

Figure 2: Android architecture 7

Figure 3: Memory Management 8

Figure 4: Memory Management 9

Figure 5: Linux Kernel 9

Figure 6: Hardware Abstraction Layer 10

Figure 7: Local and Remote IPC 10

Figure 8: Hardware Abstraction Layer 11

Figure 9: Libraries 12

Figure 10: Libraries 13

Figure 11: Object oriented Framework 14

Figure 12: Model View Presenter 15

Figure 13: Model View Presenter Framework 16

Figure 14: Started Service 18

Figure 15: Bound Service 19

Figure 16: Hybrid Service 20

Figure 17: Service 20

Figure 18: IntentService 21

Figure 19: Override 21

Figure 20: Factory Method 22

Figure 21: Android Manifest 22

Figure 22: Service Lifecycle States 1 Figure 23: Service Lifecycle States 2 23

Figure 24: Service Lifecycle Operations 24

Figure 25: Android Service 25

Figure 26: Start Service Figure 27: Bind Service 25

Figure 28: Integrating an Service into an App 26

Figure 29: Motivation for service concurrency 26

Figure 30: Service Concurrency 27

Figure 31: Intent Service Framework 27

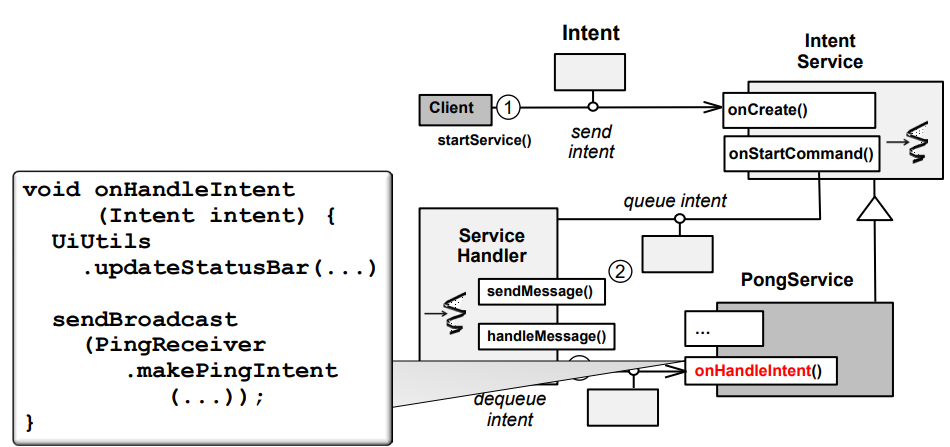
 Figure 32: IntentService Framework Structure & Functionality 28

Figure 33: IntentService Usage Considerations 29

Figure 34: Android Service Deployment Models 1 30

Figure 35: Android Service Deployment Models 2 30

Figure 36: Android Service Deployment Models 3 31

# Layered architecture

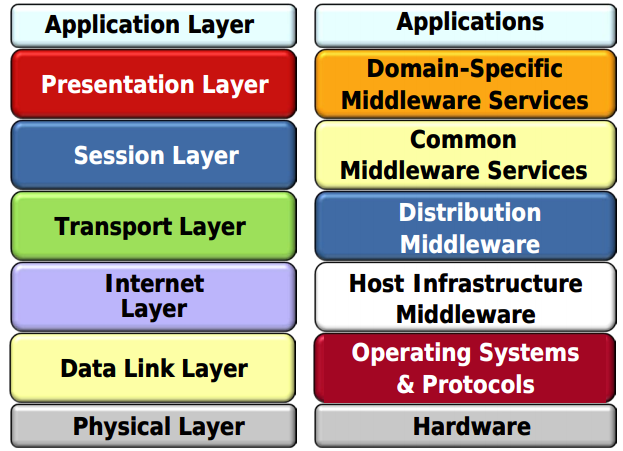


Figure 1: Layered architecture

Layering is an idea and a lot of procedures that is applied in numerous areas, going from delectable treats to antiquated structural designing activities. These protocol stacks enable end-to-end communication.

# Android Layered Architecture Overview

There are mainly 6 layers in Android architecture namely Linux Kernel, Hardware Abstraction Layer, Libraries and Android Run time, Application Framework and Applications.

Lower layer will handle interaction with hardware middle layers exchange packets between routers and hosts. The upper layers will interact with application and implement it.

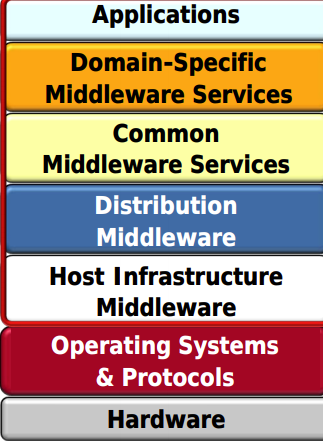
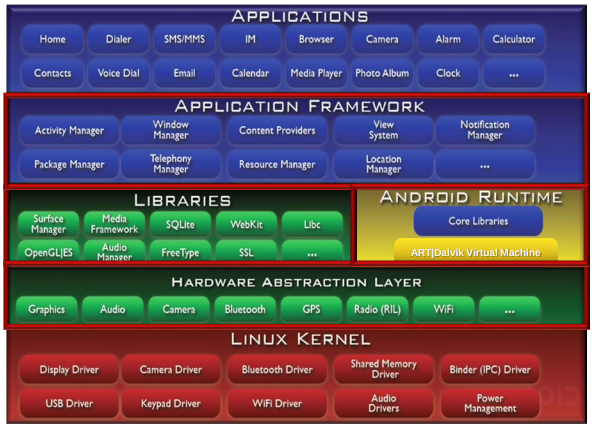
****

Figure 2: Android architecture

# Linux Kernel

## Memory management:

There are two types of storage primary and secondary storage in Linux kernel. The Kernel process operates on instructions and associated data is present inside the RAM. The data persistence is done by Secondary storage.

All Android application is executed on RAM of system. User process can access kernel by system calls. User space is more restrictive than Kernel space.

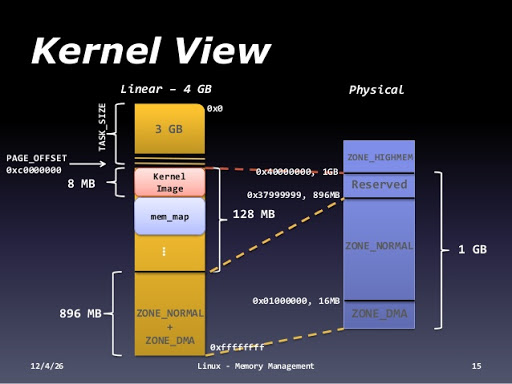


Figure 3: Memory Management

# Process Management:

Linux supports a number of Inter-Process Communication (IPC) mechanisms. Kernel Processes communicate with each other and with the kernel to coordinate their activities.

There are two types of IPCs in local and remote. This is used for interaction between System service and apps. For cloud computing IPC is essential.

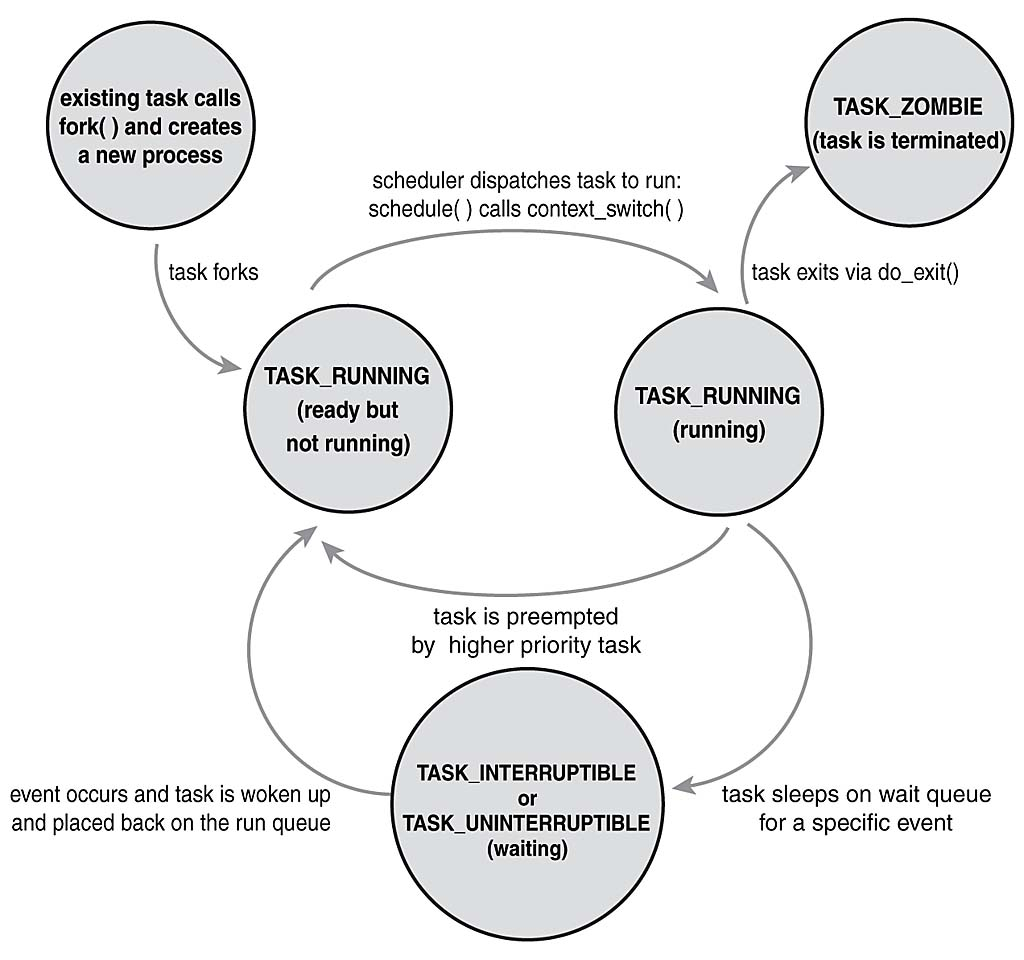


Figure 4: Memory Management

## **Kernel Extension:**

A kernel extension is a directory containing several files. App developers must know Android Linux kernels extensions for storage. Kernel extension that allows multiple processes to share regions of memory, which would otherwise be prohibited because each application process and the virtual memory manager collaborate to protect the app's memory by default.



Figure 5: Linux Kernel

# Android Hardware Abstraction Layer:

The Hardware Abstraction Layer helps to separate concerns in the Android system architecture. It separates hardware logic from android software logic.



Figure 6: Hardware Abstraction Layer

# Local and Remote IPC

Local & remote inter-process communication (IPC) mechanisms mediate interactions between apps & system services • TCP/IP is used to access the Internet • Optimized for LANs & WANs

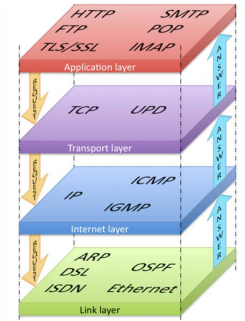


Figure 7: Local and Remote IPC

**Android Hardware Abstraction Layer**

The Android Hardware Abstraction Layer (HAL) is an interface for hardware vendors to implement that allows the Android application/framework to communicate with hardware-specific device drivers. The Android application uses HAL APIs to get service from hardware devices.

Android HAL uses the functions provided by the lower-layer Linux kernel to serve the request from the Android application/framework. Android HAL is a vendor-specific layer implemented in the C/C++ language. The HAL implementation is hardware-specific and varies from vendor to vendor.

Developers need a standardized approach to implement the Android HAL across vendors and hardware to reduce development time, cost and effort. What follows is a discussion of the challenges of implementing the Android HAL for audio, and a general solution for quick HAL implementation for various hardware vendors.

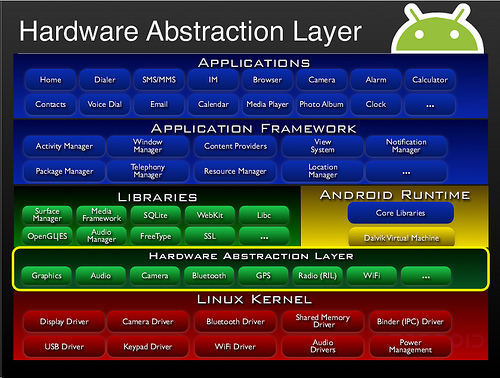


Figure 8: Hardware Abstraction Layer

# Android Libraries:

The source code for all the core Java and Android libraries is available online. Although Android apps are written using the Java APis we just discussed, implementations of these APIs are often written in C and C++.

An Android library is structurally the same as an Android app module. It can include everything needed to build an app, including source code, resource files, and an Android manifest. However, instead of compiling into an APK that runs on a device, an Android library compiles into an Android Archive (AAR) file that you can use as a dependency for an Android app module.



Figure 9: Libraries

# Android Core and Native Libraries

Here android Handlers can be used to send and process messages and runnable in one or more threads in the process. Handlers are key part of android Handle, Message and Runnable Framework.

Work Done by Handlers Include:

1. Send Message and or post runnable to threads looper

2. Collaborate with looper to serialize processing of messages in a thread

But handlers can't be used for IPC between processes and they don't implement to pierceable interface, they can't be passed as data in message and extra in an Intent.

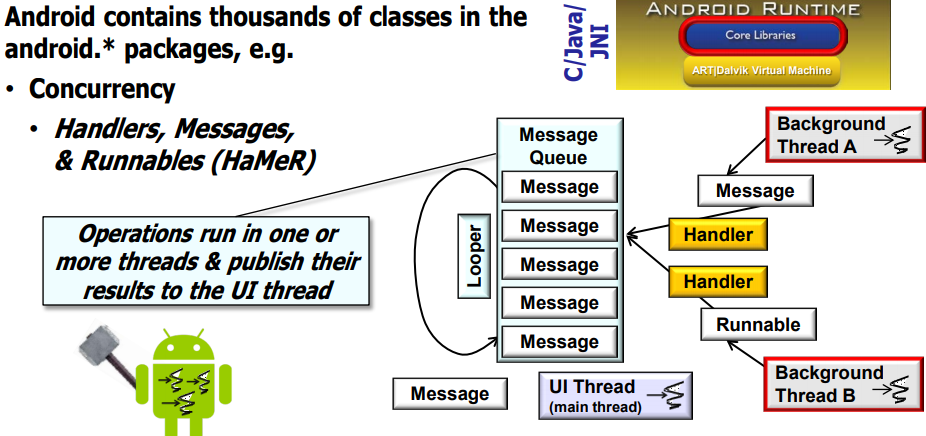


Figure 10: Libraries

# Object oriented Framework:

Object oriented framework contains all Object-oriented programming concepts.

Why Object-Oriented Framework:

* An object-oriented framework can reduce the costs of developing applications since it allows designers and implementers to reuse their previous experience on problem solving at design and code levels
* Prior research has shown that high levels of software reuse can be achieved using frameworks
* A framework is defined as a generic software for a domain (Johnson, 1997). It provides a reusable semi-finished software architecture that allows both single building blocks and the design of subsystems to be reused.

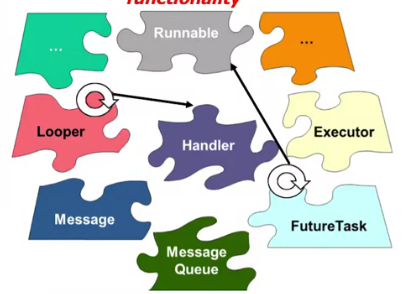


Figure 11: Object oriented Framework

# Android Services

* Activities in Android are destroyed and recreated to handle runtime configuration changes.
* Recreating an Activity
* Task and back stack

## Model View Presenter

Is a derivation of the model–view–controller (MVC) architectural pattern which mostly used for building user interfaces. In MVP, the presenter assumes the functionality of the “middle-man”. In MVP, all presentation logic is pushed to the presenter. MVP advocates separating business and persistence logic out of the Activity and Fragment. An activity processing is paused when a new activity is started and to mitigate this Model View Presenter got created.

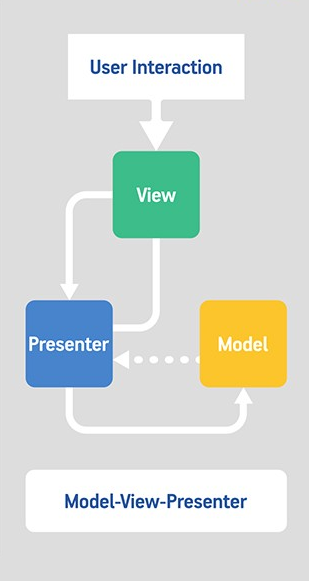


Figure 12: Model View Presenter

The Model-View-Presenter or MVP pattern can be used to alleviate some but not these limitations.

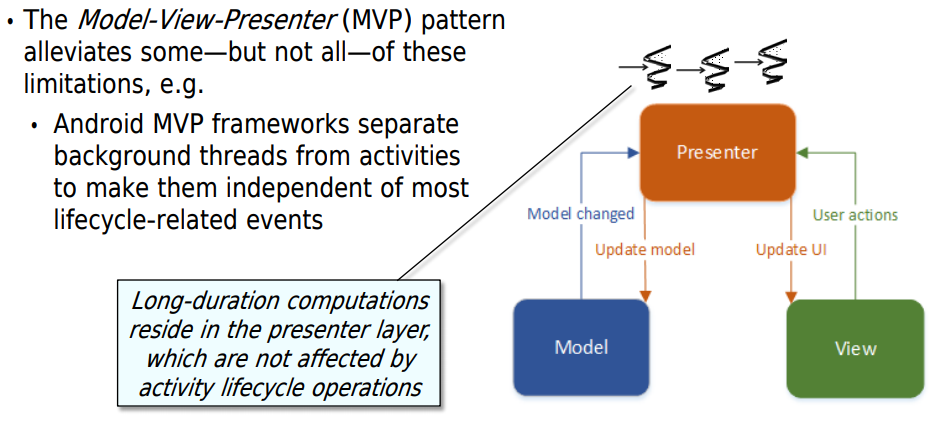


Figure 13: Model View Presenter Framework

* Typically, MVP framework implementations use so-called retained fragments to preserve key state across runtime configuration changes.
* However, MVP frameworks are not part of this standard Android code base. So, you must use third party implementations or implement your own implementation of the MVP pattern.
* Due to limitations with activities, Android, therefore, needs another type of component to perform long-duration computations in the background concurrently.
* Unlike activities, services only interact with users in very limited ways., they’re not allowed to be able to make direct calls to UI framework components.
* As a result, services must interact with the user through activities, either connected to services via local communication mechanisms that are provided by the Android binder framework. The binder framework's what's used to move the information back and forth between processes within a device,

## Services

A [Service](https://developer.android.com/reference/android/app/Service) is an [application component](https://developer.android.com/guide/components/fundamentals#Components) that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform Inter Process Communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

**Types of Android Services**

1. **Started Service**
   1. You can start a service from an activity or other application component by passing an [Intent](https://developer.android.com/reference/android/content/Intent) to [startService()](https://developer.android.com/reference/android/content/Context#startService(android.content.Intent)) or [startForegroundService()](https://developer.android.com/reference/android/content/Context" \l "startForegroundService(android.content.Intent)). The Android system calls the service's [onStartCommand()](https://developer.android.com/reference/android/app/Service#onStartCommand(android.content.Intent,%20int,%20int)) method and passes it the [Intent](https://developer.android.com/reference/android/content/Intent), which specifies which service to start. The [startService()](https://developer.android.com/reference/android/content/Context#startService(android.content.Intent)) method returns immediately, and the Android system calls the service's [onStartCommand()](https://developer.android.com/reference/android/app/Service#onStartCommand(android.content.Intent,%20int,%20int)) method. If the service isn't already running, the system first calls [onCreate()](https://developer.android.com/reference/android/app/Service" \l "onCreate()), and then it calls [onStartCommand()](https://developer.android.com/reference/android/app/Service" \l "onStartCommand(android.content.Intent,%20int,%20int)).
2. **Bound Service**
   1. A bound service is a server in a client-server interface. It allows components such as activities to bind to the services, send request, receive responses, and perform Inter Process communication. A bound service typically lives only while it serves another application component and does not run in the background indefinitely
3. **Scheduled Service**
   1. A bound service launched by a Job Scheduler that runs when certain criteria are met. Enables “batching” of services that run when the device has more resources available. Jobs Scheduler is an API for scheduling various types of jobs against the framework that will be executed in your application own process.

## Started Services

A started service is launched via startService() & has several capabilities. It’s initialized via “virtual constructor”. • Parameters can be passed via the intent used to start the service. e.g., a URL to download can be passed as “data” or a list of email addresses can be passed as “extras”. Often just performs a single operation. e.g., download an image, play a song, etc. Needn’t return a result to the client. It’s certainly possible to return a result from a started service. e.g., via a messenger or a broadcast intent. Typically shuts itself down when it’s done processing client requests. Can also be stopped by having a component call stopService().

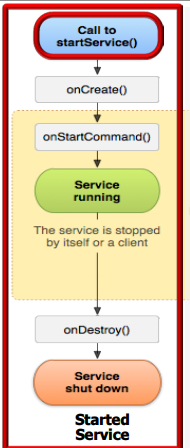


Figure 14: Started Service

## Bound Services

A bound service is launched via bindService() & has several capabilities. Performs a “handshake” between client & service to setup a connection. e.g., this connection can be used to exchange messages via IPC channels. Allow extended conversations between client(s) & the service. e.g., two-way synchronous or asynchronous messages or remote method invocations via the Binder. bindService() calls are “reference counted. A “hybrid” started/bound service is not automatically destroyed!

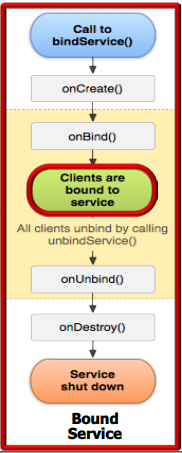


Figure 15: Bound Service

## Hybrid of Started and Bound Service

It’s possible to define “hybrid” services that combine started & bound services. If a bound service implements onStartCommand() it won’t be destroyed when it’s unbound from all clients. If onUnbind() returns “true” the onRebind() hook method will be called the next time a client binds to the service.

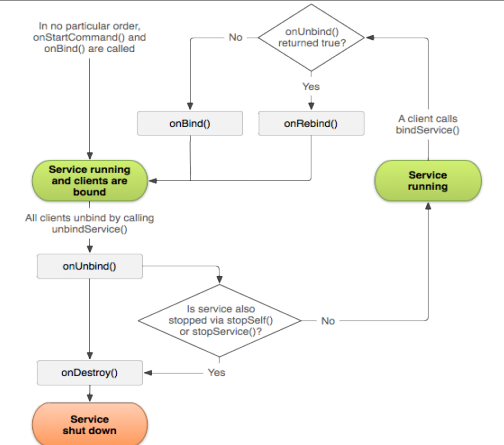


Figure 16: Hybrid Service

## Android Service Implementation

* Extend Service class, services can be extended directly with **service** keyword or indirectly with **IntentService** keyword
  1. **Services** defines a common interface for interacting with a background task, including operations performed when moving between lifecycle states.

C:\Users\TRAINING\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Service.png

Figure 17: Service

* 1. **Intent Service** Defines a base class to a framework for service concurrency that handles

C:\Users\TRAINING\AppData\Local\Microsoft\Windows\INetCache\Content.Word\IntentService.png

Figure 18: IntentService

* 1. asynchronous requests (which are expressed as intents) on demand
* Override lifecycle hook methods
  1. Subclasses define these methods to handle service state changes
  2. Android calls these methods via “inversion of control”
  3. Some lifecycle methods apply to all types of services
  4. Some lifecycle methods apply to different types of services

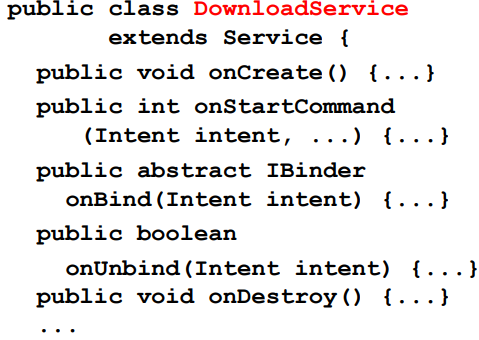


Figure 19: Override

* Define other fields, methods, & nested classes needed to implement the service
  + Factory method creates an intent used to launch service
  + Other methods & classes that implement “business logic”
  + These methods & classes also often implement a service’s concurrency & communication behaviors

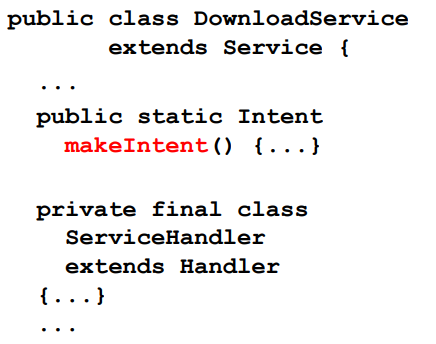


Figure 20: Factory Method

* Update the AndroidManifest.xml file to include the service so that Android knows about it
  + If services aren’t exported in AndroidManifest.xml they won’t be accessible to other apps

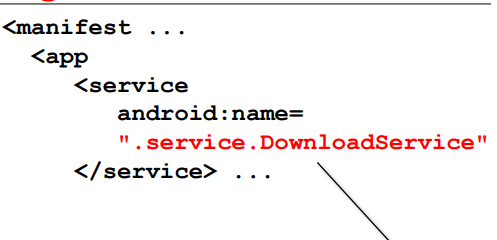


Figure 21: Android Manifest

## Service Lifecycle States

As with activities, service behavior can be expressed as a sequence of events occurring in multiple states.

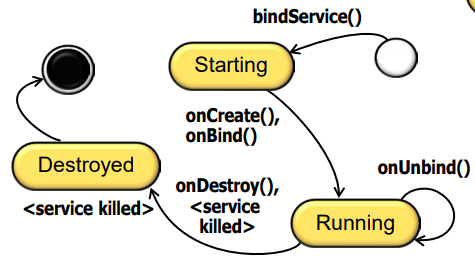
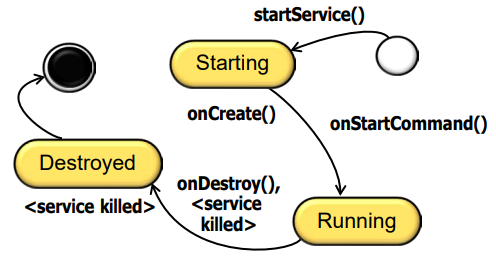


Figure 22: Service Lifecycle States 1 Figure 23: Service Lifecycle States 2

## Service Lifecycle Operations

State changes are communicated to a service via its lifecycle hook methods

**onCreate()**

1. A “virtual constructor” called after service is first launched
2. Typically used to initialize the service
3. Might also be a “no-op” if there’s nothing to initialize here

**onStartCommand()**

* + Called each time a started service is sent an intent via startService()
  + Receives the intent passed by the client’s call to startService()
  + Often used in conjunction with the service’s concurrency model
  + Returns a flag indicating semantics to use if service crashes or is killed
  + Will be a “no-op” for IntentService

**onBind()**

* + Factory method called when a client connects via bindService()
  + Receives the intent passed by the client’s call to bindService()
  + Returns a reference to a binder object to the client for later use
  + Must return null for a started service

**onUnbind()**

* + This method is called when all clients have disconnected
  + Returns “false” by default
  + Return “true” to enable new clients to bind to it later via reBind()
  + Allows binding to a started service!

**onDestroy()**

* + A “virtual destructor” disposal method called when a service is shut down
  + Destroying a started service is harder than destroying a bound service

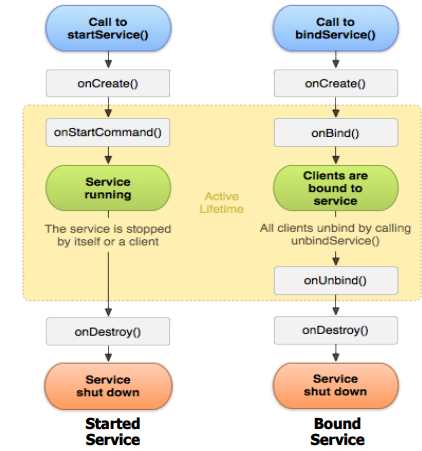


Figure 24: Service Lifecycle Operations

## Starting an Android Service

* startService() passes intent to Android Activity Manager Service
* Activity Manager Service passes the explicit intent to the component that is registered to handle it
* Activity Manager Service creates/starts a process containing the MusicService class component
* Control then passes to the MusicService’s onCreate() hook method

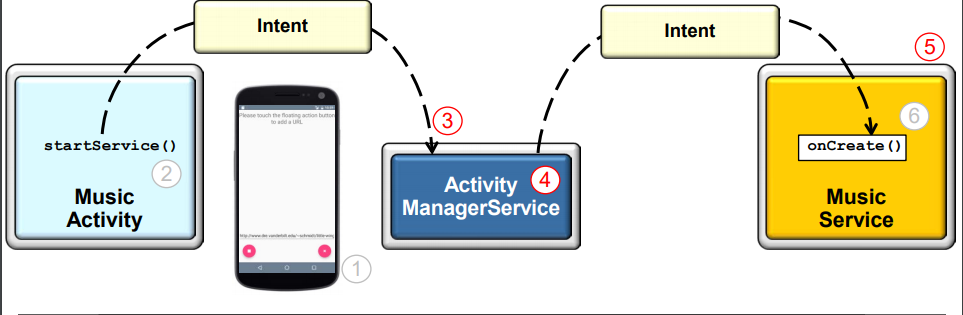


Figure 25: Android Service

## Methods used to start Android Service

• bindService() and startService()

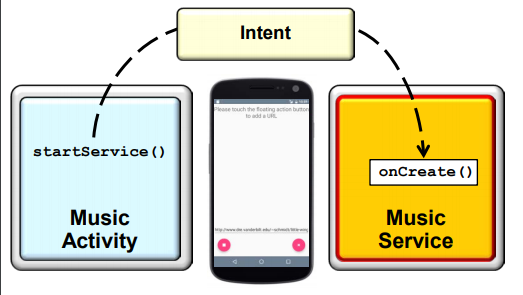
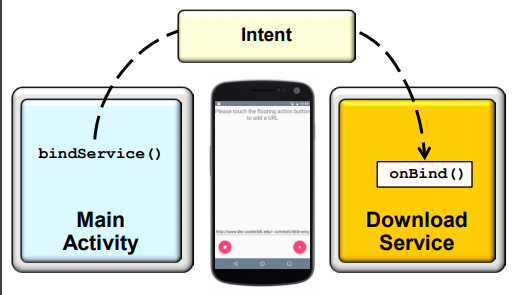
 

Figure 26: Start Service Figure 27: Bind Service

## Integrating a service into an App

* + Include the service in the AndroidManifest.xml file
  + Add element as a child of element
  + Provide the android:name to reference the service class
  + Use android:process to run a service in a separate process
  + By default, a service does not run in a separate process or thread

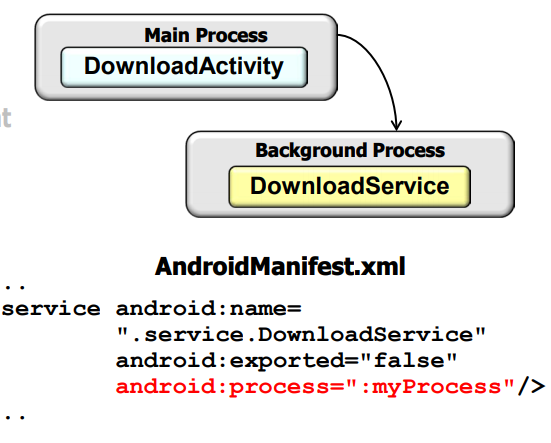


Figure 28: Integrating an Service into an App

## Concurrent Programming with Services

**Motivation**: By default, a started service runs in the main thread of its hosting process. It does not create its own thread & does not run in a separate process. A service performing long-duration or blocking operations may therefore incur Application Not Responding (ANR) errors.

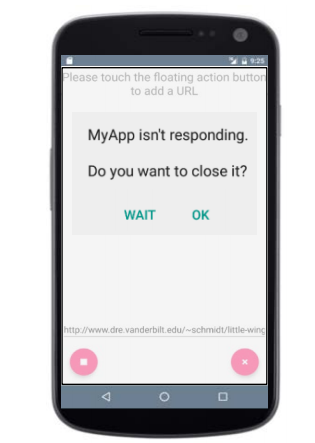


Figure 29: Motivation for service concurrency

One way to make a service concurrent is to use different processes. This configuration is controlled by AndroidManifest.xml. However, creating a process per service can use excessive amounts of system resources.

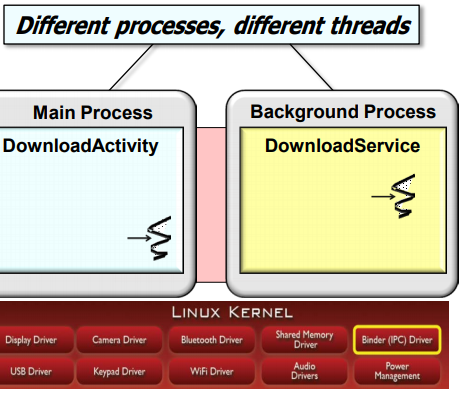


Figure 30: Service Concurrency

## Intent Service Framework structure and functionality

A multi-threaded service must be programmed rather than configured. • e.g., start a Handler Thread, execute a task in the Java Executor framework, etc. Developing ad hoc concurrent services can be tedious & error-prone. We therefore need some way to automate & simplify the programming of concurrent started services. The IntentService framework is an easy way to create a concurrent service.

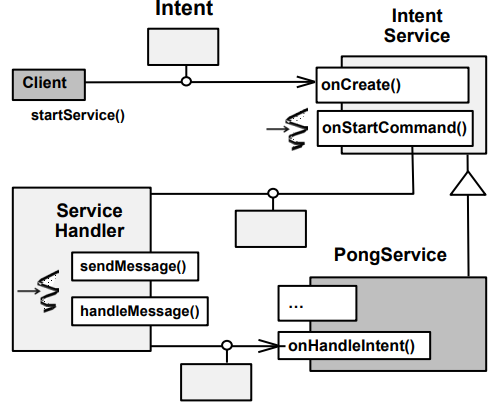
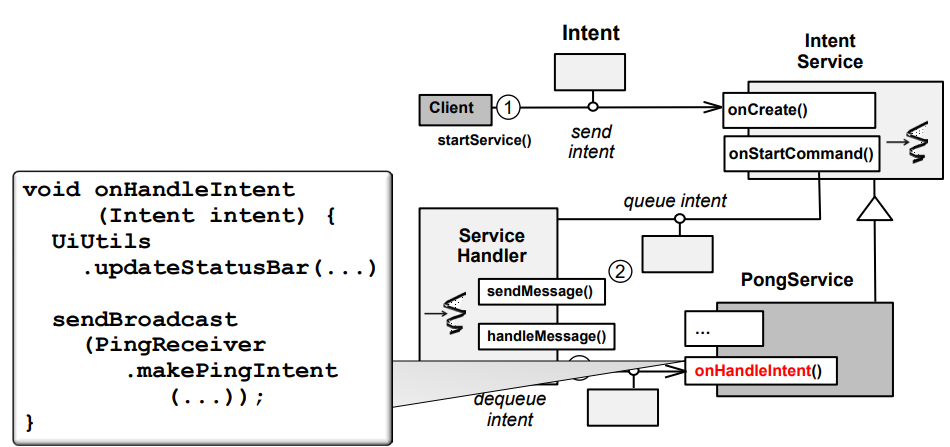


Figure 31: Intent Service Framework

Clients send intents via calls to startService(). Data can be passed with the intent. e.g., via setData() or by putting “extras” into intent. IntentService is started on-demand by the Activator pattern. An IntentService subclass implements the onHandleIntent() hook method. It processes the intent in a Handler Thread. This thread is different from the main thread

HaMeR framework dispatches onHandleIntent() hook method. The PongService app overrides onHandleIntent() to do two things. 1. Update the status bar 2. Send intent to PingReceiver.

 Figure 32: IntentService Framework Structure & Functionality

## IntentService Usage Considerations

Simplifies concurrent handling of commands pass as intents. Just override onHandleIntent() to process an intent concurrently. No need to handle runtime configuration changes. android:process can be used to can a service to run in another process. However, using a separate process may be overkill.

A key IntentService drawback is that only one intent can be processed concurrently. This limits scalability on multi-core platforms. Services that use thread pools are more scalable, but harder to shut down.

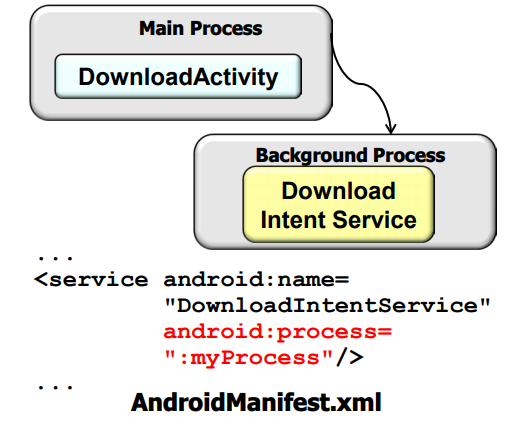


Figure 33: IntentService Usage Considerations

## Android Service Deployment Models

A deployment model directs the physical deployment of services to processes. Started & bound services can run in the same or different processes as their client components. This choice is determined via an AndroidManifest.xml file setting. The default behavior is to run a service in the same process.

An app with a service configured to run in a separate process is necessarily concurrently. Each process has a separate thread of control by default. There are several reasons for running a service in its own process, e.g. A service that’s shared by more than one app needs to run in a separate process. Running a service in its own address space can make apps more robust. Garbage collection in a separate service process doesn’t affect other processes in an app.

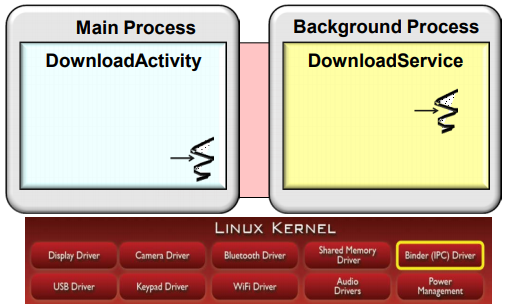


Figure 34: Android Service Deployment Models 1

## Android Service Communication Models

IPC mechanisms are needed to communicate with services in different processes & threads. Android’s Binder framework underlies its various IPC mechanisms, e.g. Asynchronous message passing via messengers & handlers. Synchronous & asynchronous remote method invocations via the Android Interface Definition Language (AIDL).

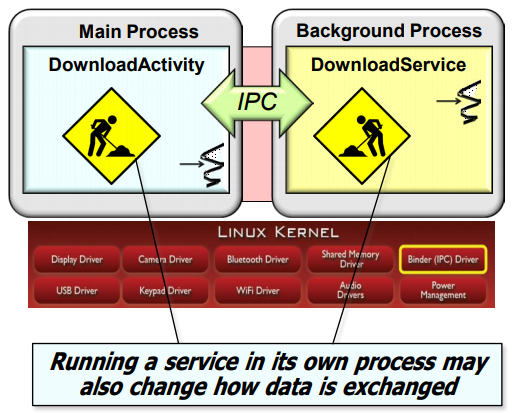


Figure 35: Android Service Deployment Models 2

IPC mechanisms are needed to communicate with services in different processes & threads. Android’s Binder framework underlies its various IPC mechanisms. Services (& activities) can implement a wide range of concurrency models. e.g., thread pool, thread-perrequest, command processor, active object, etc.

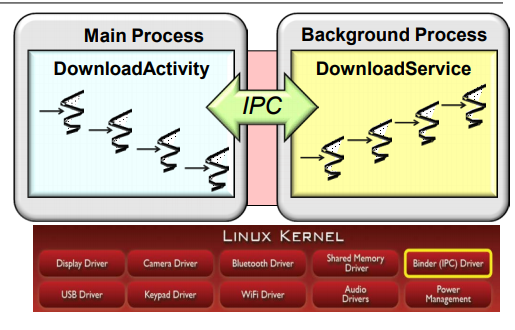


Figure 36: Android Service Deployment Models 3

## References:

<https://www.coursera.org/learn/androidapps-2/lecture/vUWDc/case-study-pingpongreceiverservice-app>