Appunti MACC

Argomenti

- Device Reference Frame and Device Orientation
    - There are three ways to express device orientation wrt ENU (East North Up)
        - Orientation angles
            - Yaw
            - Pitch
            - Roll
        - Orientation matrix
            - TRIAD algorithm
            - How to get angles from orientation matrix
        - Axis-angle
            - Singularities in 0 and pi => Quaternions
                - Efficiency
                - Android Rotation vector
                    - Android Game Rotation Vector (no magnetometer)
- Sensors
    - Transducer
    - Divide in:
        - Hardware Sensors
        - Software Sensors
    - Divide in:
        - Motion sensors
        - Environmental sensors
        - Position sensors
    - Attributes:
        - Range
        - Precision
        - Accuracy
        - Resolution
        - Sampling rate
    - Examples
        - Accelerometer
        - Gyroscope
        - Magnetometer
        - Proximity sensor

        - Ambient light sensor
        - Barometer
        - GPS
        - Fingerprint
        - Microphone
- Cloud Computing
    - Warehouse-Scale Computers (WSC)
        - Hierarchically connected
        - Crossbar-switch
        - Omega Network
    - Three levels of software
        - Server level
        - Cluster Level
        - Application level
    - Deployment model
        - Public cloud
        - Private cloud
        - Community cloud
        - Hybrid cloud
    - Service delivery model
        - Infrastructure as a Service (AWS)
        - Platform as a Service (Python anywhere)
        - Software as a Service (Gmail)
    - Necessary characteristics
        - On demand self service
        - Resource pooling and multi tenancy
        - Measured Service and Service Level Agreement
        - Rapid elasticity
            - Traditional approach
            - Autonomic approach
        - Broadband access
    - Types of cloud computing
        - Cloud computing
        - Cluster Computing
        - Distributed computing
        - Grid Computing
        - Fog Computing
            - Concept of edge of the network

- Android
    - The android software stack
        - Android APIs
        - System APIs
        - Android Framework
        - System service (traducono apis)
        - Android runtime
        - HAL (Hardware Abstract Layer)
        - Native daemons
        - Linux Kernel
    - Libraries
        - ML Kit
        - OpenCV
        - ARCore
    - Three layers architecture
        - UI (or Presentation) Layer
        - Domain Layer (optional)
        - Data Layer
    - Android apps lifecycle
        - Activity
        - Service
        - Broadcast receiver
        - Callback methods
            - OnCreate()
            - OnStart
            - OnResume
            - OnPause
            - OnStop
            - OnRestart
            - OnDestroy
    - Design
        - Design Principles
            - Separation of concerns
            - Data models should be independent of the UI
            - Single Source of Truth
            - Unidirectional Data Flow (UDF)
        - Storyboard
        - Wireframe
        - Navigation graph

                - Forward navigation
                - Backward navigation
                - Lateral navigation
                    - Hamburger Menu
                    - Bottom Navigation Bar
                    - Top Navigation Bar
        - The android view system
            - Threads
                - UI thread
                - main thread
            - Views (auto-drawing rectangles)
            - Tree of views
        - Jetpack Compose
            - Composables
            - Recomposition
            - Stateless vs Stateful
            - Hoisting
        - Asynchronous programming
            - Kotlin suspend functions (Coroutines)
                - state
                    - New
                    - Runnable
                    - Running
                    - Blocked
                    - Terminated
                - scheduler
                - Dispatchers
                    - Default
                    - IO
                    - Main
- Graphics
    - Digital Image
        - Types:
            - Binary (BW)
            - Gray scale
            - Color image
        - Color depth
        - Digital image creation

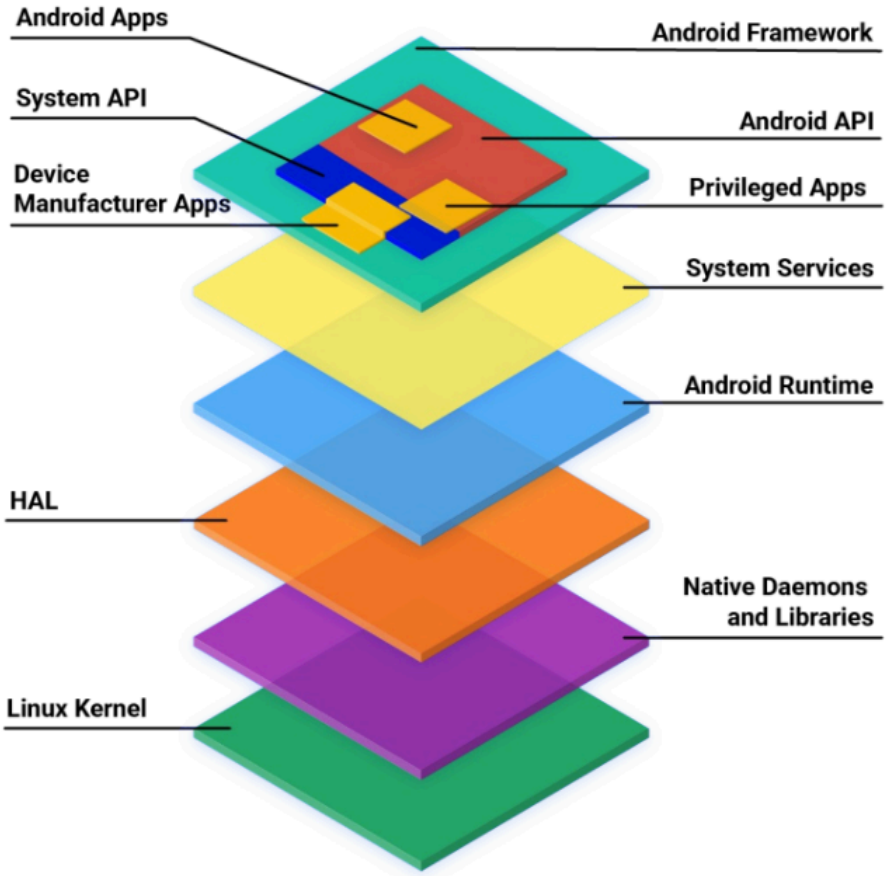
                - Sampling
                - Quantization
                - Pixel resolution
                - Spatial Resolution (PPI) with formula
                - Focal length
                - Aperture
        - Digital Screens
            - DPI vs PPI
                - Images tailored for specific DPIs in apps
            - Homogeneous coordinates
                - homogeneous reference frame
            - Transformations
                - Translation
                - Rotation
                - Scaling
                - Shear
                - Projective Transformations
                - Composition of translations and rotations
            - Basic Concepts
                - Canvas
                - Paint
                    - Color
                - Bitmaps and Drawables
                    - Texture
                    - Bitmap texture
                - Shaders
        - Augmented reality
            - Perspective projection
                - Homogeneous Coordinates(3d to 2d, 4d vectors + 1 to state it's a point and not a directory)
                    - Improper points or points at infinity
- Resource allocation
    - Markov process
        - Markov chain
        - Discrete time markov chain
        - Markov Reward Model
    - Transition probability function
        - transition probability matrix

    - Chapman-kolmogorov
    - Marginal PMF
    - Chapman–Kolmogorov equation
    - Limiting state probabilities
    - stationary or steady-state probability vector v
    - States of a chain:
        - Transient State
        - Recurrent State
            - periodic and aperiodic
            - null and nonnull
            - Mean recurrence time
        - absorbing state
    - Irreducibility
    - Irreducibility + Aperiodicity => Steady State probabilities
    - Birth-Death Markov Chains
        - Tridiagonal Matrix
    - Steady state solution:
        - Method that uses definition
        - Linear system
        - Power method
    - Balance equations
    - Continuous Time MC
        - Transition rate
    - Queue models
        - MM1
            - Traffic intensity
            - Stability condition
            - Probability of being in state k
            - Average response time
        - MMm
        - MMmn
    - Load balancing between servers
        - Query to all (expensive)
        - Random
        - Hierarchical
- Markov Decision Processes
    - Global Reward
    - Discounted Global Reward

- Average Discounted Global Reward or State value function
- Optimal State Value function
- Components of MDP
  - A MDP=(S,A,P,R, γ) where
  - S state space
  - A action space A
  - P probability transition function
    - $p(s,s',a)=Pr\{S_{t+1}=s|A_t=a_t,S_t=s'\}$, sometimes denoted as T(s,s',a)
  - R Reward function, R: SxSxA → $\mathbb{R}$,
  - γ discount factor ($0<=\gamma<1$)
- Policy
- Bellman Equation
- Best policy
  - Model based
    - Value iteration
  - Model free
    - Q-Learning

Android

Android software stack layers



- **Android apps** are created by using the Android API; they can be found by means of Google Play Store.
- **Privileged apps** are created by using a combination of the Android API and some system APIs.
- **Device manufacturer apps** are created by using a combination of the Android API, some system APIs and the direct access to the Android framework implementation.
- The **Android Framework** is a group of Java classes, interfaces and other precompiled code upon which apps are built.
- The **Android API** is a portion of the Android framework that is publicly accessible, as accessible as the SDK.
- The **System APIs** are a portion of the Android framework that is available only to the original equipment manufacturer.

- **System services** are modular, focused components that allow the functionalities exposed by the Android API to access the underlying hardware.
- The **Android Runtime (ART)** is a Java runtime environment that performs the translation of the app's bytecode into processor-specific instructions that are then executed by the device's runtime environment.
- The **Hardware Abstract Layer (HAL)** is an abstraction layer that allows Android to be agnostic about lower-level driver implementations. It has a standard interface that has to be implemented by hardware vendors.
- **Native daemons** are background processes that run in the *native code* (i.e., code compiled from languages like C or C++) on the Android operating system. These daemons directly interact with the kernel by performing low-level tasks, managing hardware or handling other critical functions.
- **Native libraries** are software libraries written in C or C++; it is possible to write native applications directly in C or C++, in order to have high efficiency, and then linking them with Kotlin code via the so-called Java Native Interface.
- The **Linux Kernel** is the central part of the operating system, and talks with the underlying hardware of the device.

Libraries

ML Kit

- ML Kit is a mobile SDK that brings Google's on-device machine learning expertise to Android and iOS apps.
- Use Vision and Natural Language APIs to solve common challenges in apps or create brand-new user experiences.
- All are powered by Google's best-in-class ML models and offered to you at no cost.
- ML Kit's APIs all run on-device, allowing for real-time use cases where you want to process a live camera stream for example. This also means that the functionality is available offline.
- ML Kit Use on-device machine learning to easily solve real-world problems, some are also available in the cloud.

OpenCV for Android

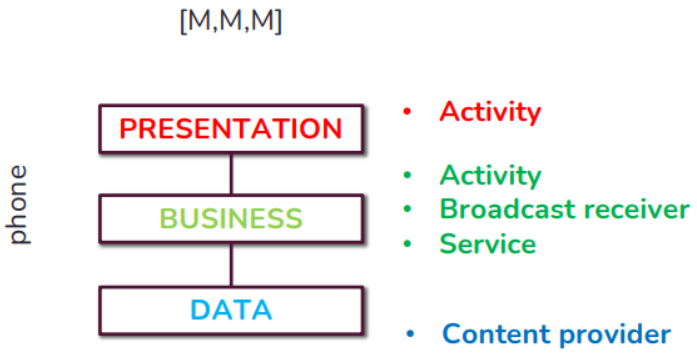
- **OpenCV (Open-Source Computer Vision Library)** is a library of programming functions mainly for real-time computer vision, including
  - Image/video I/O, processing, display (core, imgproc, highgui)
  - Object/feature detection (objdetect, features2d, nonfree)
  - Geometry-based monocular or stereo computer vision (calib3d, stitching, videostab)
  - ...
- OpenCV includes its own ML library

Arcore

- ARCore is Google's platform for building augmented reality experiences.
- Using different APIs, ARCore enables a phone to sense its environment, understand the world and interact with information. Some of the APIs are available across Android and iOS to enable shared AR experiences.
  - Motion tracking allows the phone to understand and track its position relative to the world.
  - Environmental understanding allows the phone to detect the size and location of all type of surfaces: horizontal, vertical and angled surfaces like the ground, a coffee table or walls.
  - Light estimation allows the phone to estimate the environment's current lighting conditions.

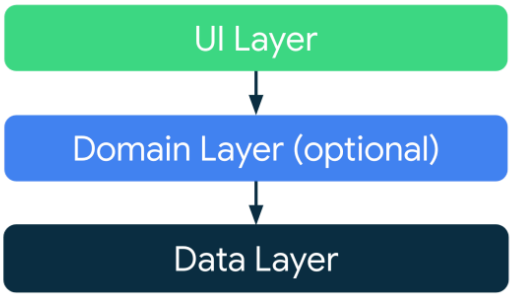
Three Layers Architecture

The following are the 5 mapping core components to the three-layers architecture



In the modern case the business part is often absent, or managed by calling external APIs:

- **UI layer** displays application data on the screen.
- **Data layer** contains (simple) business logic of the app and exposes application data.
- **Domain layer** is an optional layer that sits between the UI and data layers, responsible for encapsulating *complex business logic*.

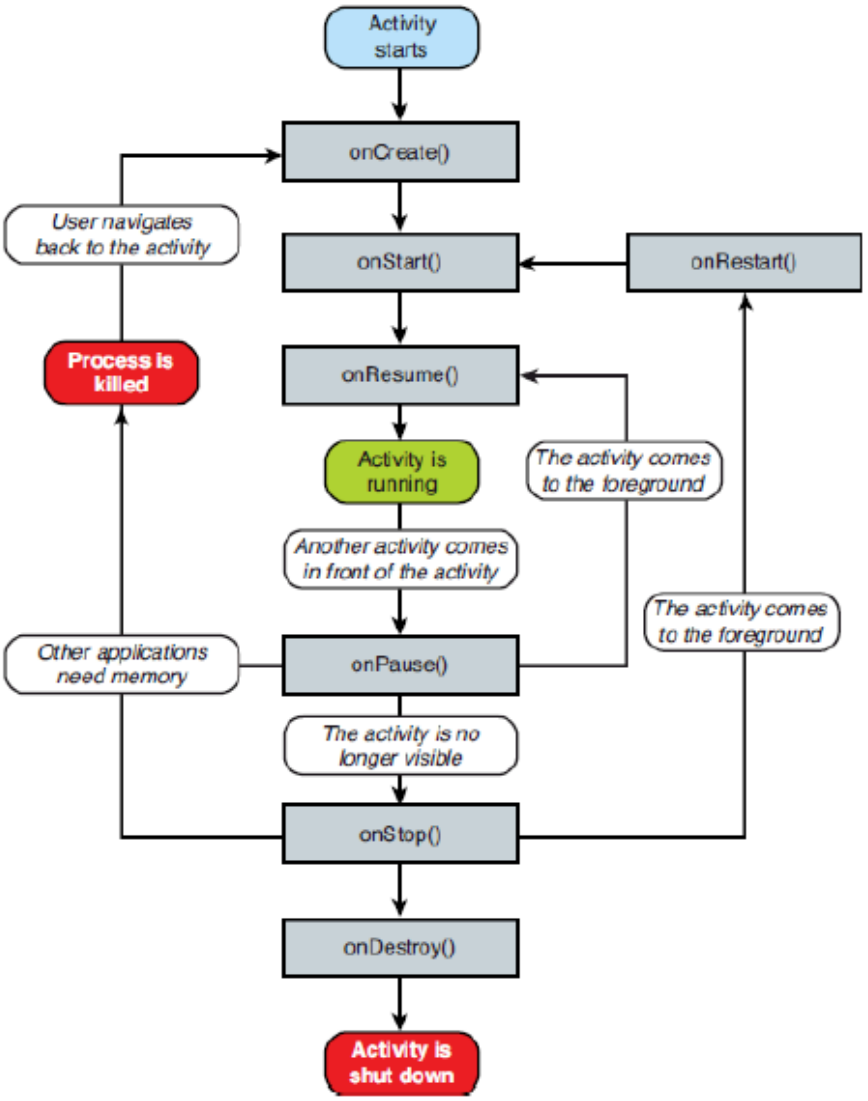


Lifecycle

A typical Android application consists of multiple components like the following:

- **Activity**: single screen with a user interface, that is responsible for interacting with the user and handling the UI components, such as buttons, text fields and images. Multiple activities can be combined for creating a complete application, and they can be transitioned between in order to provide a seamless user experience.
- **Service**: component that runs in background to perform long-running operations without a user interface. Services can run independently of the UI, and they can continue to run even if the user switches to another application (e.g., music player).
- **Broadcast receiver**: component that responds to system-wide broadcast announcements or intents. They enable the communication between Android components, and allow applications to respond to system events, such as the battery getting low or a new SMS message received.
- **Content provider**: mechanism for managing and sharing structured data between applications. This mechanism allows an application to access the data of another application, typically for enabling data sharing or accessing databases.

An activity component extends the Activity class, and must respond to a well-defined set of callback methods that are activated by the operating system (at least to the method *OnCreate()*). Globally, these callback methods define the **lifecycle** of the application.



Each activity in an application goes through its own lifecycle; it responds to a set of methods called by Android. When an activity is created, the method *onCreate()* is executed; the *onDestroy()* method is invoked when the activity that is currently running needs to be killed. In between, various methods are called, allowing the activity to be managed.

The events that trigger these callbacks, such as onCreate, onPause, and onStop, are fundamental to logically implementing the various phases of the activity. For example, the creation event (onCreate) indicates that the activity is about to be initialized, while the onPause event could be triggered by another activity becoming active, moving our activity into a paused state.

A critical aspect to consider is saving data when an activity pauses. Since the RAM dedicated to the app may be emptied during this process, you need to save data appropriately to avoid losing crucial information. This can be done through the

implementation of the onSaveInstanceState method, which allows you to preserve the state of the activity before it is paused.

Ensuring you accurately manage the activity lifecycle is essential to ensuring a smooth user experience and preventing memory management and data loss issues.

Notice that when working with Jetpack Compose, the recommended architecture is to have a single activity with multiple composable functions for different screens. Each composable function corresponds to a screen or part of the UI.

Design Principles

- **Separation of concerns.** Split the code in separate focused sections for example, UI-based classes should only contain logic that handles UI and operating system interactions and not business logic.
- **Drive UI from data models.** drive UI elements from data models. Data models represent the data of an app and are *independent* from the UI elements and other components in your app. They they are not tied to the UI and app component lifecycle but will still be destroyed when the OS decides to remove the app's process from memory.
- **Single source of truth.** assign a Single Source of Truth (SSOT) to data. The SSOT is the *owner* of that data, and only the SSOT can modify or mutate it. To achieve this, the SSOT exposes the data using an immutable type, and to modify the data, the SSOT exposes functions or receive events that other types can call.
- **Unidirectional Data Flow** (UDF) pattern. In UDF, a state (data values) flows in only one direction (from the SSOT to the UI element). The events that modify the data flow in the opposite direction.

These principles are implemented in various components, including the **jetpack architecture** components:

- Data Binding : Declaratively bind UI elements
- Lifecycles : Manages activity and fragment lifecycles
- LiveData : Notify views of any database changes
- Navigation : Handle everything needed for in-app navigation
- Paging : Gradually load information on demand from your data source
- Room : Fluent SQLite database access
- ViewModel : Manage UI-related data in a lifecycle-conscious way
- WorkManager : Manage every background jobs in Android with the circumstances we choose

When it comes to developing an Android app, it's essential to start with the design phase, which can include storyboards, mockups, and wireframes. These steps help you visualize the look and flow of your app before you start coding.

Storyboard

Creating storyboards is useful for planning the user interface structure and flow of your application. This step can help clearly define how users will interact with the app in terms of navigating between various screens.

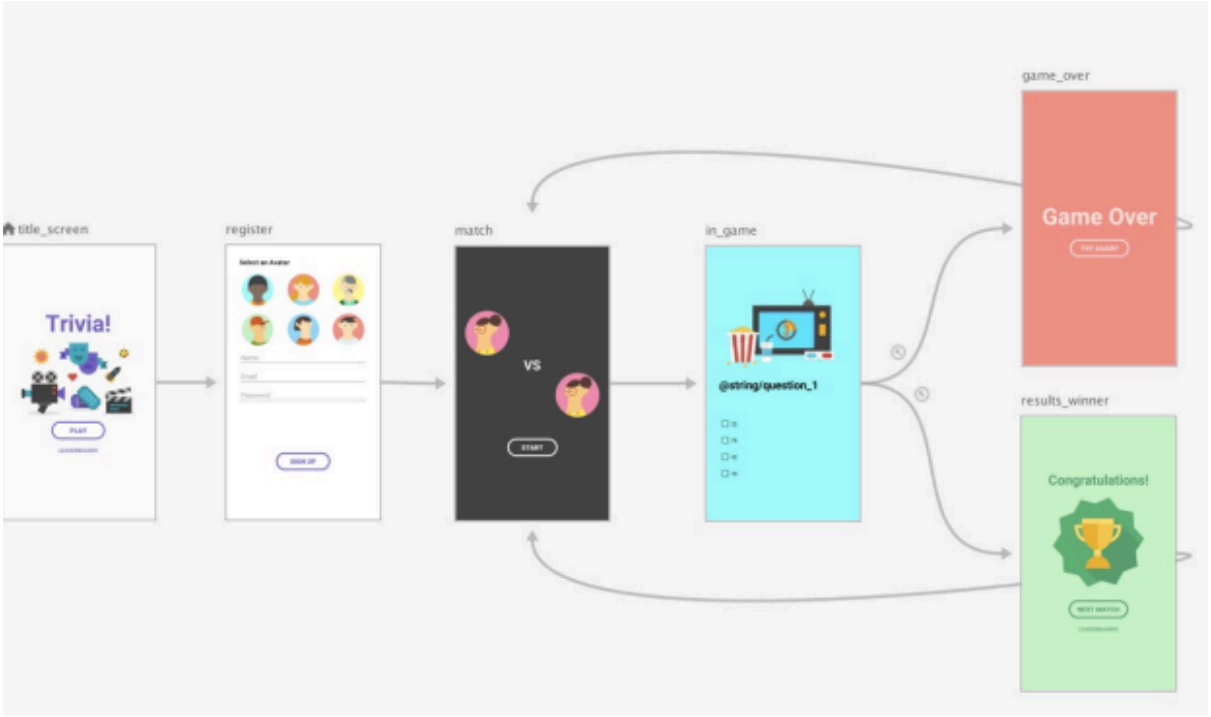
Mockup and Wireframes

Mockups and wireframes are visual tools that allow you to represent your user interface layout and design in more detail. Balsamiq and Figma are great tools for this purpose. While Balsamiq focuses on more static wireframes, Figma offers a collaborative environment that allows you to create interactive prototypes. You can use these tools to get timely feedback on your design before moving into the actual development phase.

Navigation Graph in Android

The Navigation Graph is a key component when designing an Android app using the navigation architecture recommended by Android Jetpack. It graphically represents the structure and navigation of the app, showing how the different screens are connected to each other. This visual tool makes it easier to understand the flow of your application and helps you efficiently manage navigation between different tasks or fragments.

You can use Android Studio to create and view the Navigation Graph, making it easier to manage your app's navigation paths. Be sure to clearly define navigation actions, entry points, and destinations in your navigation graph to ensure smooth and intuitive navigation for your app users.



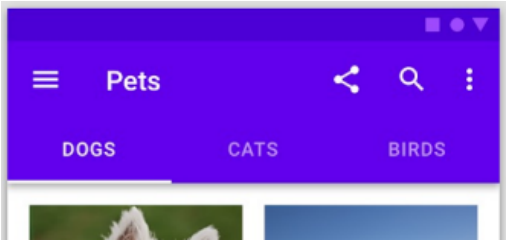
Navigation Patterns

There are different types of navigation in Android development. **Forward navigation** occurs when the user moves deeper into the application, while **backward navigation** occurs when the user moves back in the hierarchy (if the user is at the top of the hierarchy, this can result in exiting from the application). **Lateral navigation** occurs when you navigate between two pages in the same hierarchy (for example, from a red page to a green page through a link).

Regarding lateral navigation, there are different standards. Some include:

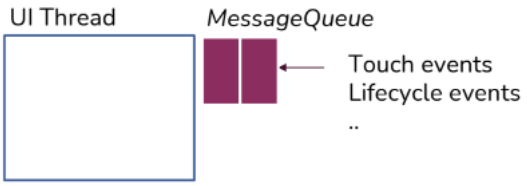
- **Navigation Drawer** (**Hamburger Menu**): a hamburger icon in the top left that, when clicked, opens a side menu containing navigation options.
- **Bottom Navigation Bar**: a navigation bar located at the bottom of the app, often with icons representing different sections of the application (as seen in apps like Spotify and Instagram).
- **Top Navigation Bar**: a navigation bar at the top of the application, which can contain icons or text to allow the user to navigate between different sections.

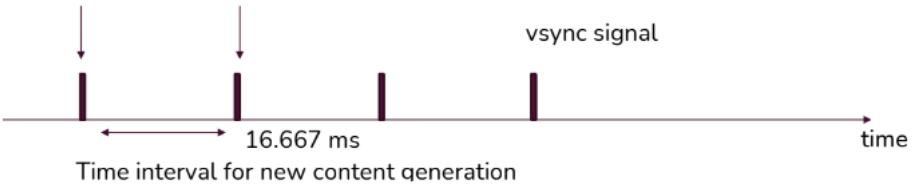
Each approach has its advantages and can be chosen based on your specific application needs and design preferences. For example, the Navigation Drawer is often used to hide less common options or advanced navigation spaces, while the Bottom Navigation Bar is better suited for apps with a few main sections and a clear hierarchy. The choice also depends on consistency with Android design guidelines and user expectations.



The android view system

- Let's now consider some technical issues of how Android generates screen content
- The Activity code runs inside a thread, called **UI thread** or **main thread**
  - An app is made of many threads
- This code includes the definition of the screen content
- Screen content are updated every 1/60 = 16.7 ms, which provides a refresh rate of 60 Frame per Second (FPS)



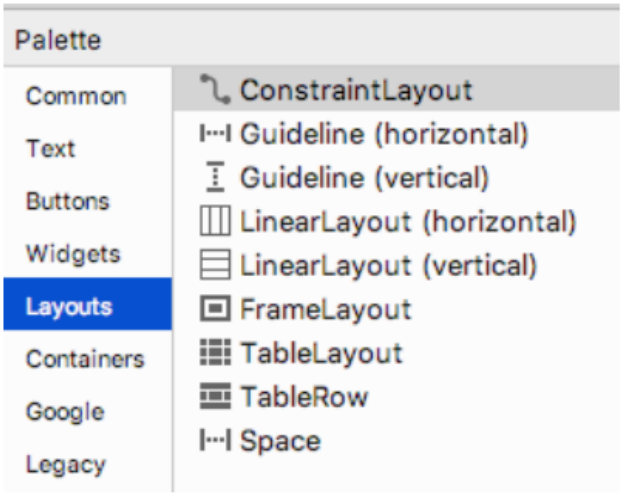


- The area of the screen that can change is considered as a **tree of views**
- **View** = rectangular shape on the screen that 'knows how to draw itself' wrt to the containing view
- View can be populated using

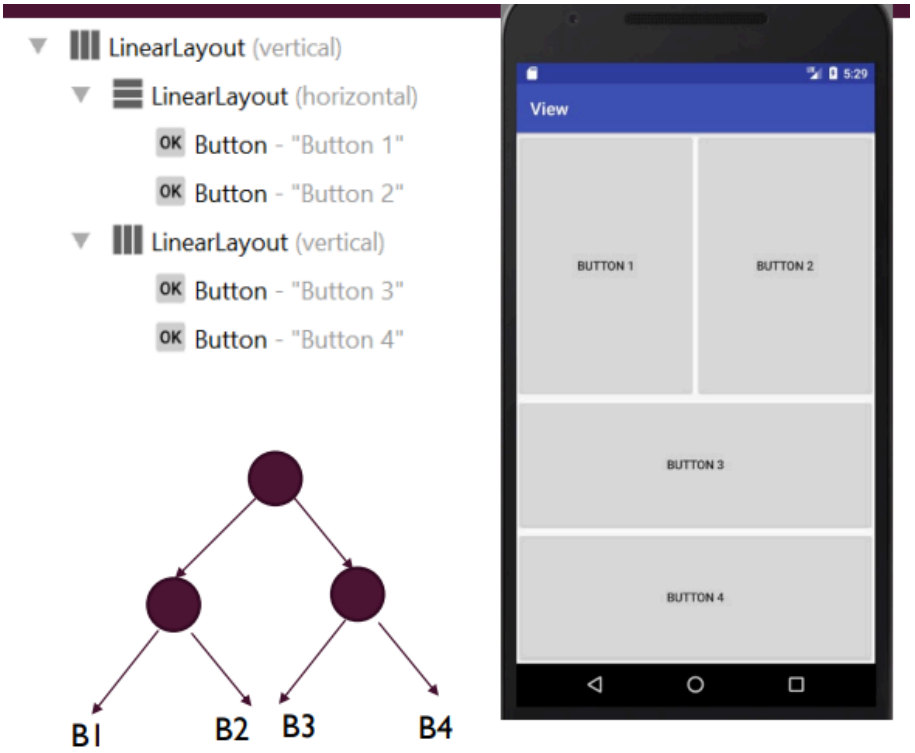
1. UI toolkits in the Android framework are based on:
   - Imperative language (the 'old' way) using XML files describing UI building blocks, like the HTML philosophy
   - Declarative language using composable functions
2. Draw directly via Canvas abstractions (2D graphics or openGL ES)

When developing an Android app, it's common to organize the user interface using a hierarchy of nested views. In imperative mode, in the onCreate() method, you usually call setContentView(layout) to specify the main layout of your activity. However, when using Android Studio, it is often preferred to use the declarative approach, which involves dragging and dropping into the visual layout editor.

In the process of creating the user interface with Android Studio, you can drag and drop Views into the layout through the visual editor. For example, you could start by creating a ConstraintLayout and then add the desired views into it.



I can build more complicated things by inserting layout into layout:

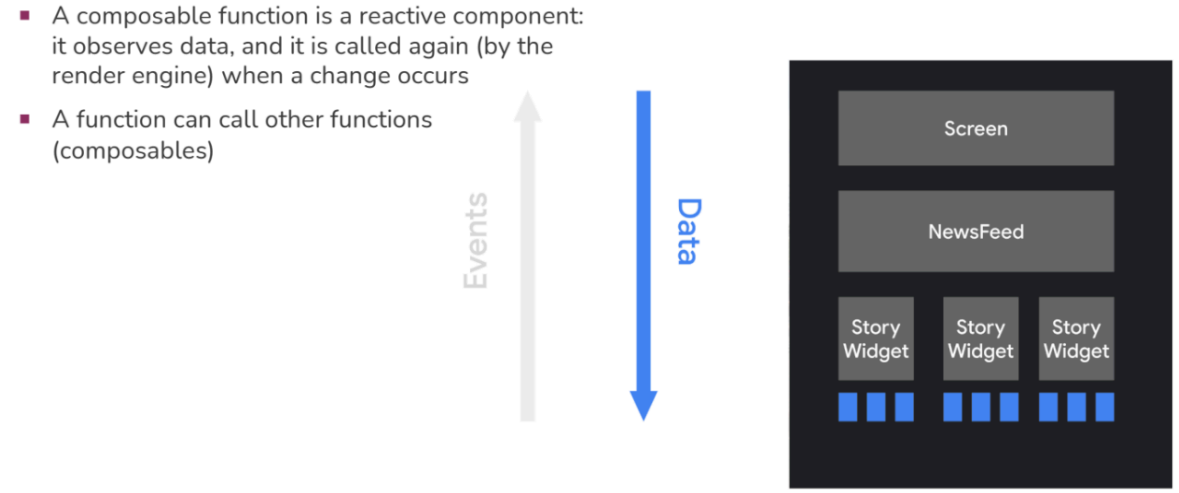


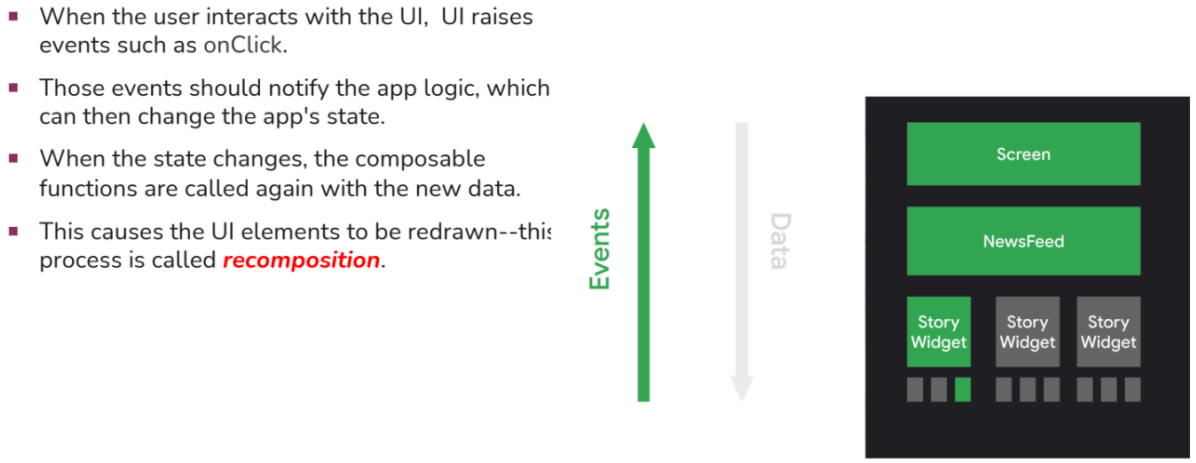
Containers are dedicated to elements that do not fit entirely on the screen and require the ability to scroll. One of these is the RecyclerView, a container optimized to display lists of items efficiently, ensuring smooth performance even when items go out of view on the screen.

Jetpack Compose

Jetpack Compose is a library used to create user interfaces quickly. Two key concepts of Jetpack Compose are "**composable functions**" and the **reactive approach**. A composable function represents a reactive component that is called whenever the associated data changes. This function can in turn call other composable functions, creating a structure similar to the reactive programming model.

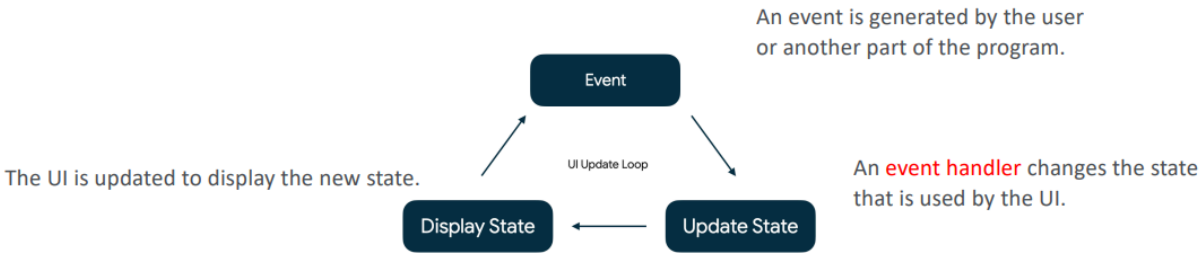
When a user interaction occurs, the UI raises an onClick event, which can be used to interact with application logic. This interaction can lead to a change of state, which, in turn, will trigger a process known as "**recomposition**". During recomposition, the involved composable functions are redrawn to reflect updates in the application state.





In summary, Jetpack Compose simplifies user interface development through the use of **responsive composable functions**, automatically managing the interface update process in response to state changes or user interactions.

- The **state** of an application is any value that can change over time
- An **event** is any action that causes the modification of a state.
- Events notify a part of a program that something has happened.
- In all Android apps, there's a core UI update loop that goes like this:



The "remember" function in Android is used to store the state of a variable during redials. This is especially useful in the context of Compose, where recompositions can be frequent. The "rememberSaveable" function performs the same function as "remember", but preserves state beyond application termination, ensuring that state is restored when the application is restarted.

"mutableStateOf" is a function that indicates that the associated variable is mutable and that any change to its value will cause a recomposition. It is often used within lambda functions. For example, if you use "mutableStateOf" for a variable called "count", any change in the value of "count" will automatically cause the parts of the UI that depend on that value to recompose.

Density-independent pixels (dp) represent a measurement independent of pixel density, allowing for more flexible management of the size of user interface elements on displays with different pixel densities.

Hoisting

It is a common practice to keep composables as "stateless" as possible to maximize code reusability. When you need to manage state, you can adopt the technique of "**hoisting**", which consists of elevating state management to a separate function. This approach helps keep composables lighter and more presentation-focused, while state management is

delegated elsewhere.

Here's a basic example:

kotlin Copy code

```kotlin
// A stateless Counter composable
@Composable
fun Counter(count: Int, onIncrement: () -> Unit) {
    Column {
        Text("Count: $count")
        Button(onClick = onIncrement) {
            Text("Increment")
        }
    }
}

// The parent composable where the state is managed
@Composable
fun ParentComposable() {
    var count by remember { mutableStateOf(0) }

    Counter(count = count, onIncrement = { count++ })
}
```

A **stateless** composable is a composable that doesn't own any state, meaning it doesn't hold or define or modify new state.

A **stateful** composable is a composable that owns a piece of state that can change over time.

In real apps, having a 100% stateless composable can be difficult to achieve depending on the composable's responsibilities. You should design your composables in a way that they will own as little state as possible and allow the state to be hoisted, when it makes sense, by exposing it in the composable's API.

In this example:

1. The `Counter` composable takes `count` (the current count) and `onIncrement` (a function to increment the count) as parameters. It does not manage any state itself.
2. The `ParentComposable` manages the state. It has a `count` state that is passed down to `Counter`. It also passes a lambda to increment the count, which `Counter` will call when the button is clicked.
3. This makes `Counter` easily reusable and testable, as it does not depend on any specific state management logic.

- A composable function is a Kotlin function that takes data as input and emits UI elements
- Functions are composable because they can call other functions
- Eventually, there are primitive functions that correspond to basic UI building blocks (or widgets), like a text view

Kotlin

The Kotlin language, used for Android development, shares some similarities with Java, but introduces concepts that make code more concise and flexible. A distinctive feature is the presence of functional programming, which allows you to pass functions as arguments or assign them to variables. Additionally, to prevent null pointer exceptions, we encourage you to avoid having variables set to null unless you explicitly force the variable to do so. Another interesting feature is the use of open classes, which allows the addition of methods and attributes to a class without the need to create a subclass.

Kotlin also introduces the distinction between mutable variables (using 'var') and immutable variables (using 'val'). You can define default values for function parameters, for example: fun name(ParName: type = defaultValue).

Lambda functions are represented in the format {input1, input2 → output}, offering a compact syntax for expressing anonymous functions. To use GitHub in Android Studio, simply click "Get from VCS".

It is recommended to use API 28, and Gradle plays an important role as a plugin for dependency management and Android project configuration. When creating a new project, it is recommended to select "Empty Views Activity" to ensure the presence of the layout folder.

The project manifesto specifies the activity that serves as the system entry point, within the <intent-filter>. In case of multiple activities, it is necessary to modify the manifesto. Message exchanges between tasks are called "intents".

2D Graphics

There are three types of images

- **Binary image** pixel can be 0 or 1, either on or off
- **Gray image** where the value of each pixel represents the total amount of light
- **Color image** the value of a pixel is a triple (red, green and blue), **RGB** color model

Basic Concepts

Mesh

In computer graphics, a **mesh** refers to a collection of vertices, edges, and faces that defines the shape of a polyhedral object in 3D space. Meshes are fundamental components in 3D graphics, used to create and represent various objects in virtual environments.

Shader

A **shader** is a program or script that runs on the GPU (Graphics Processing Unit) and is used to manipulate the appearance of objects in a 3D environment. Shaders are commonly employed to control aspects such as color, lighting, and shading effects to achieve realistic or stylized visuals. They can be categorized into vertex shaders (dealing with geometry and vertices) and fragment/pixel shaders (dealing with individual pixels and colors).

Texture

A **texture** is an image or a data array that is applied to the surface of a 3D model to enhance its visual appearance. Textures are used to simulate surface details, colors, and patterns, providing a more realistic or stylized look to objects. They can be 2D images or procedural textures generated algorithmically.

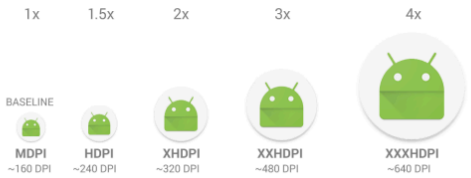
Bitmap Texture

A bitmap texture is a specific type of texture that is based on a bitmap image. Bitmap images are made up of pixels, where each pixel stores color information. These textures are often used for more detailed and realistic surface representation, as they can capture intricate patterns and details.

DPI

The resolution of digital screens is expressed by Dots Per Inch (DPI)

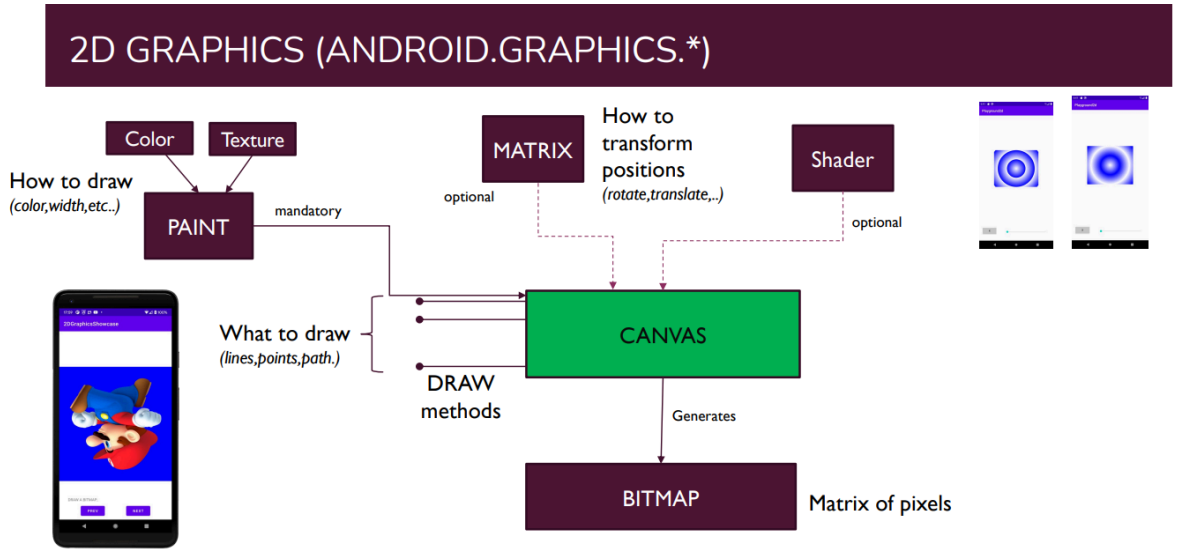
- Consider an image of 2400 x 3000 pixels (7.2 MP)
- On a 300 DPI screen, the image is large 2400/300 x 3000/300=8''x10'' (20,32 cm x 25,4 cm)
- If the DPI changes (decreases), the same image is bigger...
- the higher the density the shorter the physical size of a same bitmap on the screen
- As an app usually targets many different devices, one then needs to use images with DPI dependent size
- Android defines 5 standard classes for device resolution



Android 2D graphics

Android 2D graphics is the set of activities that comprehends rendering and manipulating two-dimensional graphics. This includes displaying images, drawing shapes, handling user input, and creating interactive user interfaces. Android provides a robust set of tools and APIs for working with 2D graphics, making it possible to create visually appealing and responsive applications.

Basic concepts



Canvas

The Canvas class in Android provides a drawing surface for 2D graphics. It allows you to draw shapes, text, and bitmaps on the screen. You can obtain a Canvas object for a View or a Bitmap and use various methods to draw graphics elements.

Paint

The Paint class is used to define how to draw graphics elements on the Canvas. It includes attributes such as color, style, stroke width, and text size. Developers can customize the appearance of shapes, text, and other graphical elements by setting properties in the Paint object.

Bitmaps and Drawables

Android uses bitmaps to represent images. The Bitmap class allows for the loading and manipulation of image data. Drawables are a higher-level abstraction that can represent various types of visual elements, including images, shapes, and gradients.

Views and ViewGroups

In Android, the UI is built using View and ViewGroup components. Views are individual UI elements (buttons, text fields, etc.), while ViewGroups are containers that hold multiple Views. Custom drawing on a View is often done by extending the View class and overriding the onDraw method.

Animation

Android supports 2D animations to create visually appealing and interactive user interfaces. Animations can be applied to Views, drawables, and other graphical elements. The

Animation framework provides classes for handling animations, including translations, rotations, fades, and more.

Touch Handling

Android provides mechanisms to handle touch input, allowing users to interact with 2D graphics. Touch events can be captured and used to trigger specific actions or movements.

OpenGL ES

For more advanced graphics, Android supports OpenGL ES (Embedded Systems), which is a cross-platform API for 2D and 3D graphics rendering. It allows for high-performance graphics rendering using the GPU.

Developers can leverage these concepts and tools to create visually engaging and responsive 2D applications on the Android platform, ranging from simple games to sophisticated user interfaces. The Android SDK provides documentation and sample code to assist developers in implementing 2D graphics effectively.

Homogeneous coordinates

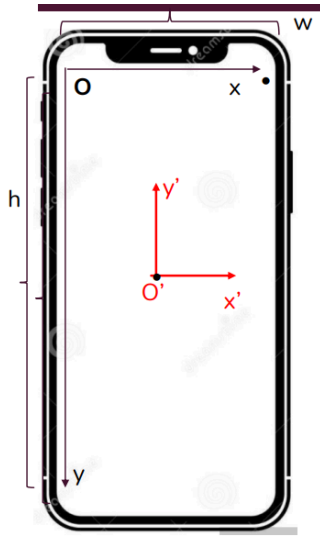
Homogeneous coordinates are a system of coordinates used in projective geometry and computer graphics. They provide a way of representing points in a space of N dimensions using $N + 1$ coordinates, which is particularly useful for representing points at infinity and for performing geometric transformations.

Here's a basic overview of how they work and their applications:

1. **Representation**: In homogeneous coordinates, a point in N-dimensional space is represented by $N + 1$ coordinates, not just N as in Cartesian coordinates. For example, a point in 2D space (Cartesian coordinates x, y) is represented in homogeneous coordinates as (wx, wy, w), where w is a non-zero scalar. Similarly, a point in 3D space (Cartesian coordinates x, y, z) is represented as (wx, wy, wz, w).
2. **Points at Infinity**: Homogeneous coordinates are particularly useful for representing points at infinity, which are important in projective geometry. In this system, points at infinity are represented by setting $w = 0$. For example, the point at infinity in the direction of the x-axis in 2D space is represented as $(1, 0, 0)$.
3. **Geometric Transformations**: Homogeneous coordinates simplify the mathematics of geometric transformations such as translation, rotation, and scaling. In traditional Cartesian coordinates, these transformations might require different sets of equations or the use of matrices plus additional translation vectors. In contrast, with homogeneous coordinates, all these transformations can be represented uniformly as matrix multiplications, allowing for more efficient and simpler computation.
4. **Computer Graphics and Computer Vision**: In computer graphics, homogeneous coordinates enable the easy description of camera transformations and perspective projections. They allow for the construction of the view matrix and projection matrix used in the rendering pipeline. Similarly, in computer vision, they facilitate the description of camera models and the transformation of images.
5. **Conversion to Cartesian Coordinates**: To convert back to Cartesian coordinates from homogeneous coordinates, you divide each component by w. For instance, the homogeneous coordinates (wx, wy, w) correspond to the Cartesian coordinates $(x/w, y/w)$.

- The homogenous coordinates in a plane are triples (x,y,w) obeying to the following rules:
- (x,y,0) represents a **direction** (or **vector**)
  - Any λ(x,y,0) is an equivalent representation of the same direction (any λ<>0)
  - (-x,-y,0) has the same direction of (x,y,0) but opposite **orientation**
- (x,y,1) represents a **point**
  - Any λ(x,y, λ) is an equivalent representation of the point
  - Or (x,y,w) represents the point (x/w,y/w,1)
- (0,0,0) is not legal

Homogeneous Reference Frame



- **REF'** is the **D**evice reference frame, **REF** the **S**creen drawing reference
- To know the coordinates of a point in **REF**, starting from those in **REF'**, we must define the coordinates of
- vector **x**': (1,0,0)
- vector **y**': (0,-1,0) (orientation is swapped)
- Origin **O**': (w/2,h/2,1)

- The change-of-basis matrix is then:
- ${}^{S}\mathbf{M}_{D}=\begin{pmatrix}1 & 0 & w/2\\ 0 & -1 & h/2\\ 0 & 0 & 1\end{pmatrix}$

Transformation Matrices: Transformations like translation, rotation, scaling, and perspective projection are represented as matrices in the homogeneous coordinate system. These matrices can be combined (multiplied) to create complex transformations. The advantage of using homogeneous coordinates is that all these transformations can be represented uniformly as matrix operations, including translation, which cannot be represented as a matrix multiplication in Cartesian coordinates.

Application in 3D Graphics: In 3D graphics, the homogeneous reference frame is fundamental. It allows for a consistent and efficient way to describe the position and orientation of objects, the camera view transformation, and the projection transformation (including perspective projection). The entire rendering pipeline in modern graphics systems, like OpenGL or DirectX, leverages this approach.

Transformation name	Affine matrix	Example
Identity (transform to original image)	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Translation	$\begin{bmatrix} 1 & 0 & v_x > 0 \\ 0 & 1 & v_y = 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Reflection	$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Reflection	$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Scale	$\begin{bmatrix} c_x = 2 & 0 & 0 \\ 0 & c_y = 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	
Rotate	$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	 where $\theta = \frac{\pi}{6} = 30°$
Shear	$\begin{bmatrix} 1 & c_x = 0.5 & 0 \\ c_y = 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	

In addition to the transformations mentioned above, there are so-called **projective transformations** that maintain straight lines, but not necessarily parallels. This type of transformation is particularly useful when you have an image containing elements read in perspective, allowing you to manually correct the perspective.

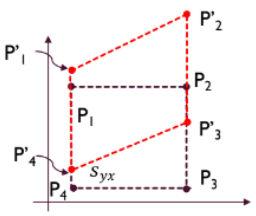
- Projective transformations have the last row with not zeros
- Without loss of generativity, the last element is one
- The are 8 independent coefficients to chose
- The transformation preserves straight lines, but not parallelism among lines

$$\mathbf{M}= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{pmatrix}$$

Space: 1999

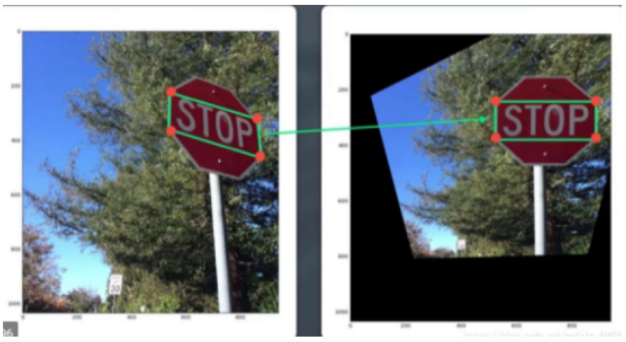
A practical example could be a photograph of a "stop" sign, where you map a rectangle around the writing and straighten it.

$$S=\begin{pmatrix} 1 & 0 & 0 \\ s_{yx} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



$y' = y + s_{yx}\, x$

Angles not preserved



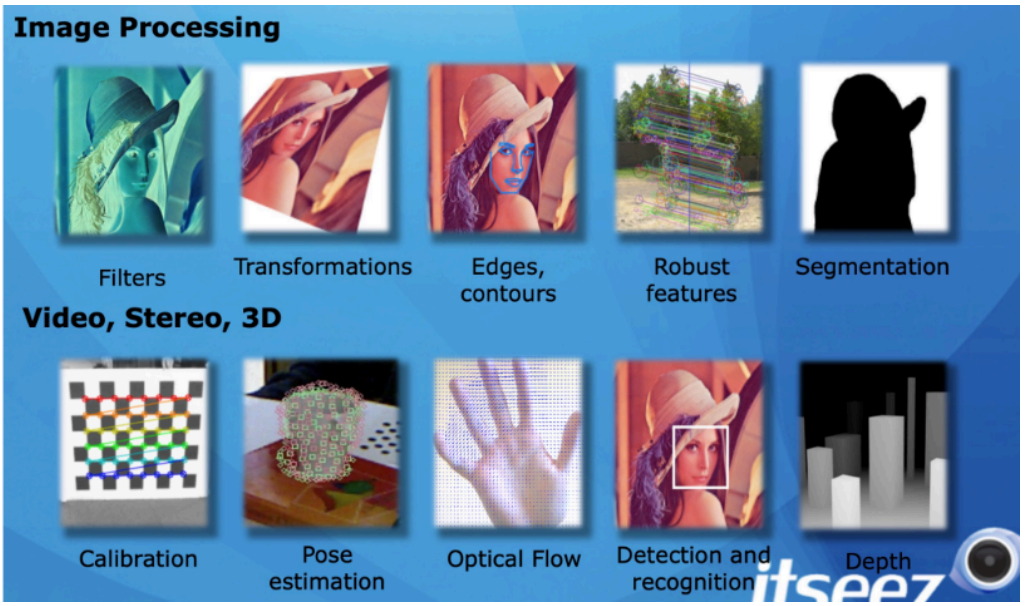
Importantly, Android supports both **Scalable Vector Graphics** (SVG) and **bitmaps** (for example, JPG images). This flexibility allows developers to use both types of graphics depending on the needs of the project. SVGs are particularly advantageous as they allow for lossless scaling, while bitmaps are ideal for detailed, complex images.

As for coloring 2D graphics in Android, it is possible to use shaders, where the colors are determined using mathematical rules (for example, circular gradient), or textures, where the color of each pixel is defined starting from an image of basic.

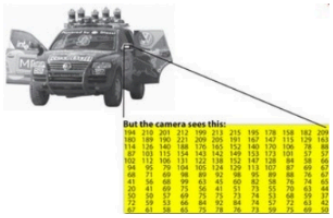
OpenCV

- OpenCV: Open Source Computer Vision Library
- OpenCV is the leading open-source library for computer vision, image processing and machine learning, with features GPU acceleration for real-time operation.
- OpenCV is released under a BSD license and hence it's free for both academic and commercial use. It has C++, C, Python and Java interfaces and supports Windows, Linux, Mac OS, iOS and Android.

Data structures in OpenCV are critical for image manipulation and processing.



MAT: THE BASIC OPENCV IMAGE CONTAINER



- *Mat* is the data structure of an image
- It is divided into two parts
- A fixed size matrix header
  - Contains meta-info such as the size of the actual data matrix, the method used for storing, at which address is the matrix stored, Point Of Interests (POI) and so on
- A pointer a to **multichannel** array, containing the pixel values
  - One channel per component (e.g, RGB)
- Two *Mat* objects can share the same matrix data
  - For example, copy just make a copy of the header
  - Clone makes a deep copy

SCALAR

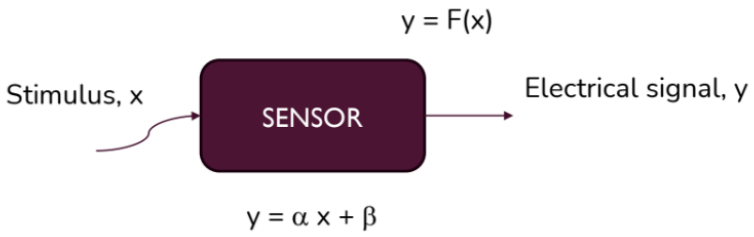
- **Scalar** is a class that represents a pixel
- A scalar can have from up to 4 components (depending on the matrix type it is used in)
  - Example: scalar.val[0] gives the first component

Sensors

- A cell phone is equipped with numerous sensors that allow it to measure external quantities, useful for making the application work in a location-aware way
- Is my face very close to the screen?
- What is the light intensity ?
- What it the external temperature?
- How is the device oriented in the space?

What is a sensor?

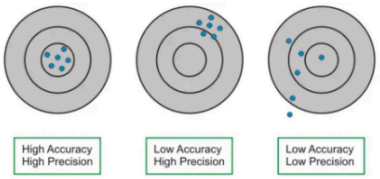
- The active element of a sensor is called a **transducer**
- A transducer is device which converts one form of energy to another
- Nowadays the other form of energy is electrical



Calibration: determine α and β

Main attributes of a sensor

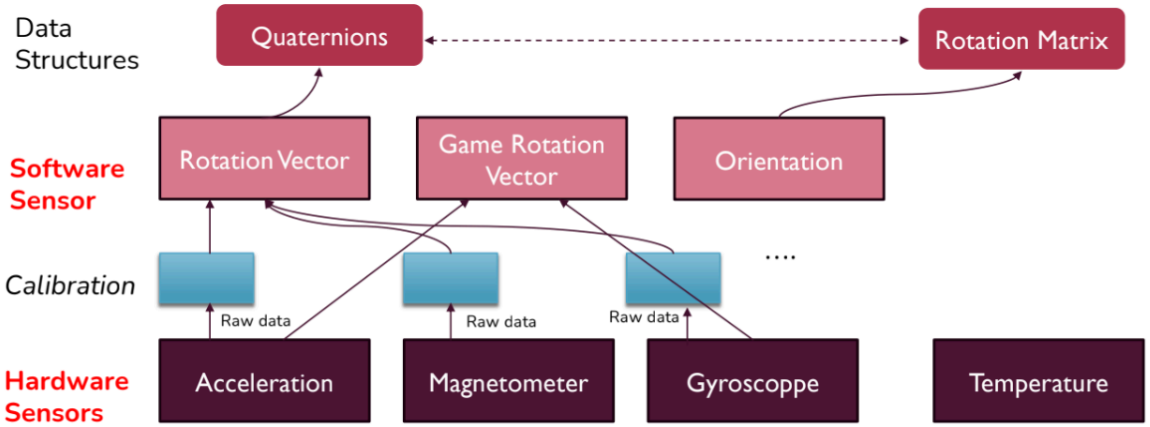
- **Range**: It is the minimum and maximum value of physical variable that the sensor can sense or measure.
- **Precision**: It is defined as the 'closeness' among a set of values concerning the same true value
- **Accuracy**: It is defined as the difference between measured value (Xm) and true value (Xt). It is defined in terms of % of full scale or % of reading
  - Absolute error = |Xt-Xm|,
  - Relative error = |Xt-Xm|/Xt
  - High Accuracy → High Precision , not vice versa
- **Resolution**: It is the minimum change in input that can be sensed
- **Sampling rate**: the frequency at which data is produced



Sensors in android

- **Hardware sensors** that correspond to physical chips, usually are Micro-Electro-Mechanical Systems (MEMS), composed of two parts:
  - Mechanical part that exploits some physical law
  - Electrical part: used to transduce the mechanical quantity into an electrical readable value (e.g., acceleration → volts)

- **Software sensors** they merge (**sensor fusion**) many data from hardware sensors to provide a more abstracted valuable information (like the orientation, as described later)



- The Android platform supports three broad categories of sensors:
- **Motion sensors**: These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes
- **Environmental sensors**: These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.
- **Position sensors**: These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

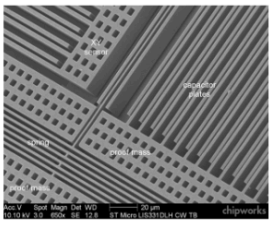
- At programming level, all sensors are managed by a **SensorManager**
- A sensor generates **events** either continuously (every ΔT s, the sampling rate), on-change (the event is generated upon a change), or one-shot
- Program register a **SensorEventListener** to **SensorEvent**

Accelerometer sensor

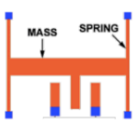
- Suppose mobile device is accelerated
- The proof mass (say M) sees three forces:

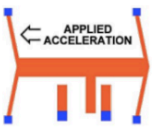
1. Force due to its inertia proportional to the external acceleration: F_E=Ma
2. Damper force proportional to the speed of the mass: F_D= D $\dot{x}$
3. Spring force proportional to the displacement of the mass: F_S=K x

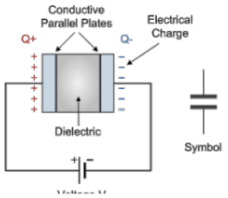
- Overall, the displacement x depends on the acceleration a, because
- Ma + D $\dot{x}$ + K x =M $\ddot{x}$
- The displacement modifies the capacity and generates a current



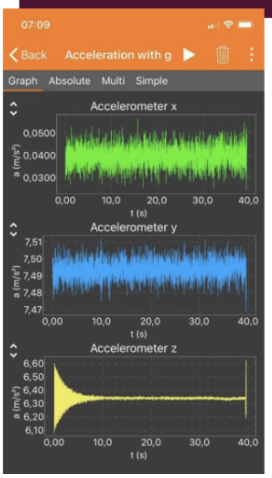
$$a = \ddot{x} + \frac{D}{M}\dot{x} + \frac{K}{M}x$$







EXAMPLE: TOUCH THE SCREEN



removed via (filtering)

vibrations from the environment

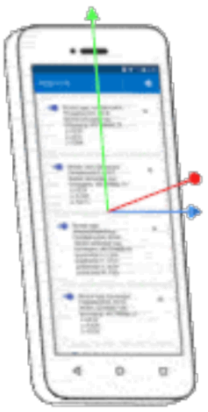
Vibrations due to screen touch (to start recording)

Gyroscope sensor

X: 0.0 rad/s

Y: 0.0 rad/s

Z: 0.0 rad/s

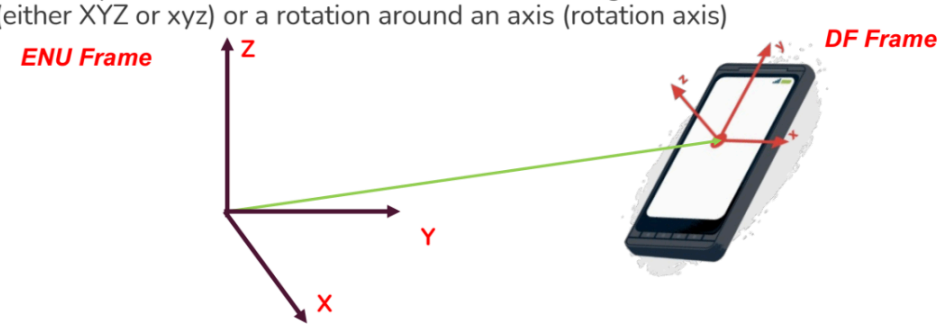


Geomagnetic sensor

Based on Hall effect, which detects the magnitude of a magnetic field through the production of a potential difference across an electrical conductor that is traversed by a magnetic field

Device Orientation

- **Dynamic definition**:
- The orientation of a phone is represented by the rotations necessary for the ENU frame (fixed frame, sometimes represented as XYX) to align with the DF frame (mobile frame, or xyz)
- Can be expressed as a sequence of **three** rotations around the orthogonal axis of the reference frames (either XYZ or xyz) or a rotation around an axis (rotation axis)



- There are three main ways to express the orientation

1. **Orientation angles**, Yaw, Pitch, Roll are the amplitude of the rotations needed to align the axis (note the sequence of the rotation axis is important)
2. **Orientation matrix** (aka rotation matrix or Direction Cosine Matrix, DCM)
3. **Axis-angle** (used by Quaternion)

- These quantities are all estimated using sensor readings from accelerometer, magnetometer, and gyroscope (in some algorithm)

Orientation Matrix

- We start with the orientation matrix, **R** which is easily estimated with only two sensors
- In Android there is a specific method ***getOrientationMatrix*** that returns the matrix by merging readings
- Let denote as $\{\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3\}$ the basis of ENU (fixed frame) and $\{\mathbf{e'}_1,\mathbf{e'}_2,\mathbf{e'}_3\}$ the basis of the DF (device frame), where instead of xyz we used 1,2,3
- To make the introduction concrete, let consider an arbitrary external magnetic field **F**, e.g. due to a magnet and let $\mathbf{m}=(m_1,m_2,m_3)$ be the measurements in $\{\mathbf{e'}_1,\mathbf{e'}_2,\mathbf{e'}_3\}$ and $\mathbf{m'}=(m'_1,m'_2,m'_3)$ the measures in the $\{\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3\}$

- The **orientation matrix** is the relationship between **m** and **m'**
- $\mathbf{F}=m'_1\mathbf{e}'_1+ m'_2\mathbf{e}'_2+ m'_3\mathbf{e}'_3 = m_1\mathbf{e}_1+ m_1\mathbf{e}_2+ m_3\mathbf{e}_3$
- If r_{ij} is the i-th component of $\mathbf{e}'_j$ in $\{\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3\}$ then
- $m'_1\mathbf{e}'_1+ m'_1\mathbf{e}'_2+ m'_3\mathbf{e}'_3 =$
- $m'_1\underbrace{[r_{11}\mathbf{e}_1 + r_{21}\mathbf{e}_2\, r_{31}\mathbf{e}_3]}_{\mathbf{e}'_1} + m'_2\underbrace{[r_{12}\mathbf{e}_1 + r_{22}\mathbf{e}_2\, r_{32}\,\mathbf{e}_3]}_{\mathbf{e}'_2} + m'_3\underbrace{[r_{13}\mathbf{e}_1 + r_{23}\mathbf{e}_2\, r_{33}\,\mathbf{e}_3]}_{\mathbf{e}'_3} =$
- $m_1\mathbf{e}_1+ m_2\mathbf{e}_2+ m_3\mathbf{e}_3$
- Where $m_1=r_{11}m'_1+r_{12}m'_2+r_{13}m'_3= \mathbf{r}_1 \cdot \mathbf{m}'$; etc..
- $\begin{bmatrix} m_1 \\ m_2 \\ m_3 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} m'_1 \\ m'_2 \\ m'_3 \end{bmatrix}$ $\qquad \mathbf{m} = {}^{ENU}\mathbf{R}_{DF}\,\mathbf{m}'$

- Column j of the matrix contains the coordinate of $\mathbf{e}'_j$ in the ENU frame
- The element r_{ij} is the coordinate of $\mathbf{e}'_j$ along direction $\mathbf{e}_i$,
- Since $r_{ij}= \mathbf{e}'_j\cdot\mathbf{e}_i = |\mathbf{e}'_j||\mathbf{e}_i| \cos(\mathbf{e}'_j\mathbf{e}_i) = \cos(\mathbf{e}'_j\mathbf{e}_i)$ the matrix is called **Direction Cosine Matrix**
- From the definition of the coefficients, ${}^{ENU}\mathbf{R}_{DF}$ is nothing more than a linear mapping from one space to another
- In our case, this mapping is interpreted as the fixed reference frame 'moved' so that it is aligned to the mobile frame

Properties

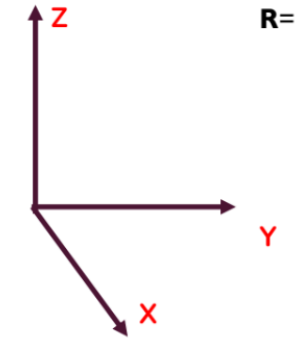
- The orientation matrix has the following properties

1. The inverse of the matrix is equal to the transpose of the matrix
2. The determinant to of the matrix is equal to 1

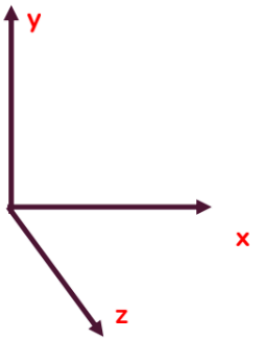
- Properties 1 **and** 2 implies the matrix is an orientation matrix

Examples

1)



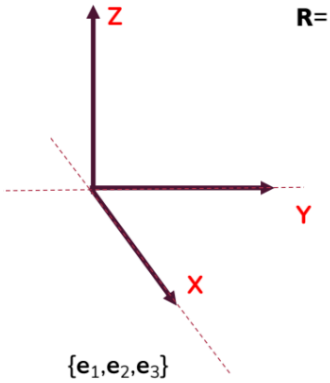
$$\mathbf{R}=\begin{bmatrix}0 & 0 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}$$



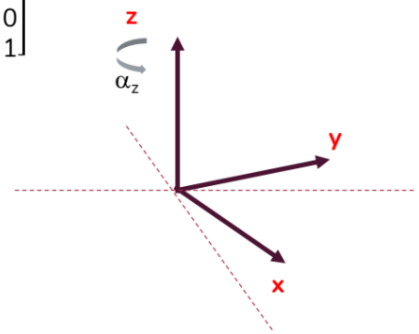
$\{\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3\}$

$\{\mathbf{e'}_1,\mathbf{e'}_2,\mathbf{e'}_3\}$

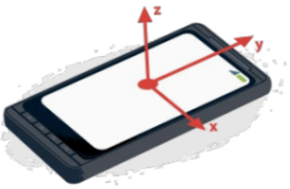
2)



$$\mathbf{R}=\begin{bmatrix}c_z & -s_z & 0\\ s_z & c_z & 0\\ 0 & 0 & 1\end{bmatrix}$$



$c_z=\cos(\alpha_z)$
$s_z=\sin(\alpha_z)$



$\{\mathbf{e}_1,\mathbf{e}_2,\mathbf{e}_3\}$

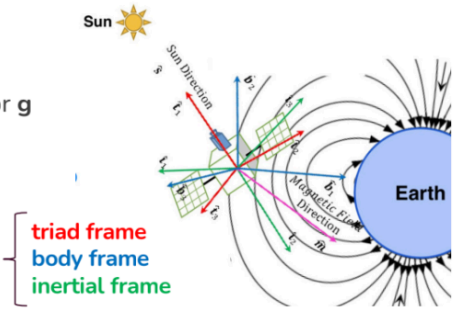
$\{\mathbf{e'}_1,\mathbf{e'}_2,\mathbf{e'}_3\}$

Triad Algorithm for Orientation estimation

- The Tri-Axial Attitude Determination (TRIAD) was first described in 1964* to estimate the attitude of spacecrafts starting from the position of the sun (using a star tracker) and the magnetic field of Earth **B**

- TRIAD uses two no-collinear vectors to define three orthogonal axis ($\mathbf{t}_1$,$\mathbf{t}_2$,$\mathbf{t}_3$) and the relationship between the two frames w.r.t. these axis

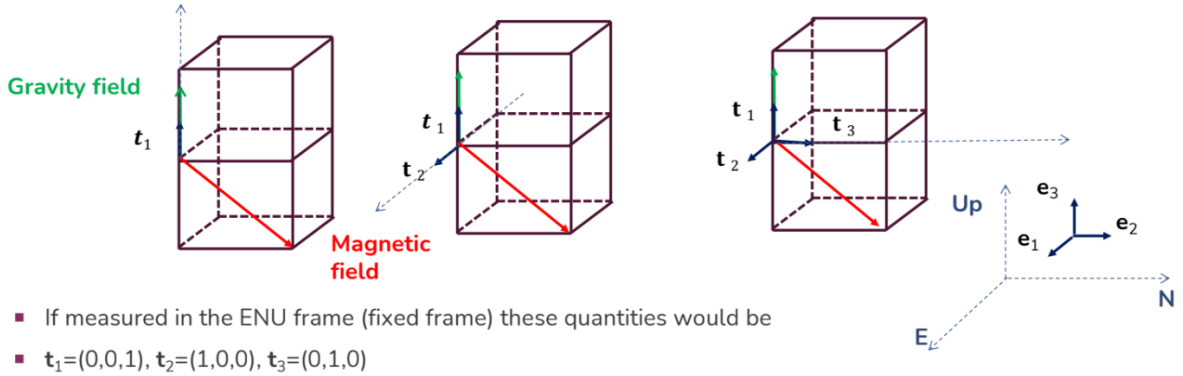
- In android, the direction of the sun is replaced with the vector **g**



(*) Black, Harold. "A Passive System for Determining the Attitude of a Satellite," AIAA Journal, Vol. 2, July 1964, pp. 1350–1351

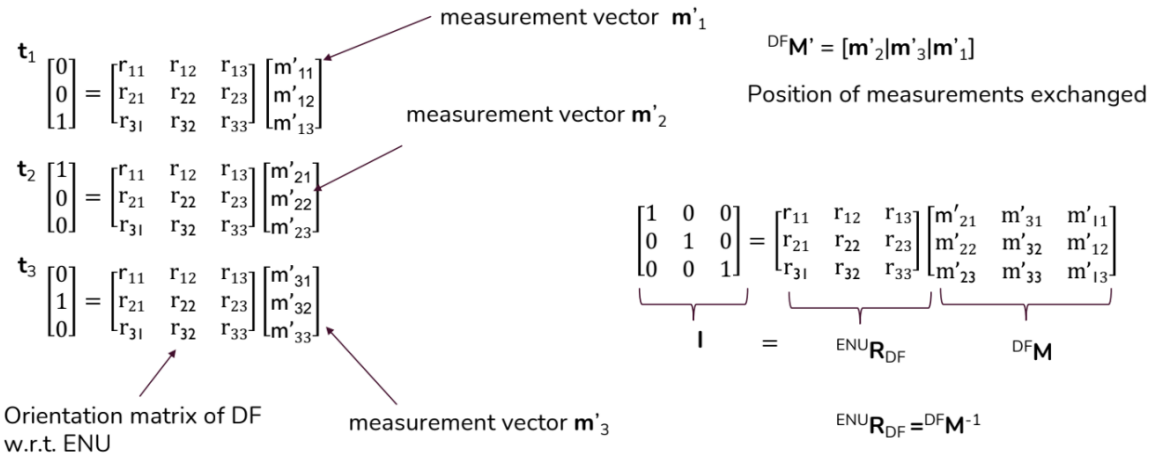
Definition of the three Orthogonal Axes

- $t_1 = \frac{\mathbf{g}}{|\mathbf{g}|}$, $t_2 = \frac{\mathbf{B} \times \mathbf{g}}{|\mathbf{B} \times \mathbf{g}|}$, $t_3 = t_1 \times t_2$



- If measured in the ENU frame (fixed frame) these quantities would be
- $\mathbf{t}_1$=(0,0,1), $\mathbf{t}_2$=(1,0,0), $\mathbf{t}_3$=(0,1,0)
- Let $\mathbf{m'}_1$ $\mathbf{m'}_2$ $\mathbf{m'}_3$, the measurement vectors of $\mathbf{t}_1$,$\mathbf{t}_2$,$\mathbf{t}_3$ the device frame

RELASHIONSHIP BETWEEN FRAMES



TRIAD ALGORITHM

- Since $\mathbf{M}^T=\mathbf{M}^{-1}$

$$ {}^{ENU}\mathbf{R}_{DF} = \begin{pmatrix} \frac{\mathbf{B}\times\mathbf{g}}{|\mathbf{B}\times\mathbf{g}|} \\ \frac{\mathbf{g}}{|\mathbf{g}|}\times\frac{\mathbf{B}\times\mathbf{g}}{|\mathbf{B}\times\mathbf{g}|} \\ \frac{\mathbf{g}}{|\mathbf{g}|} \end{pmatrix} $$

Orientation angles

- The orientation matrix **R** has only three independent parameters, and one common set of such parameters are the **orientation angles**
- These angles are defined starting from the following dynamic interpretation on the orientation matrix
- We start with the mobile frame (DF) aligned with the fixed frame (ENU). Then, the device is rotated three times along orthogonal directions of an angle with amplitude $\alpha_1,\alpha_2,\alpha_3$ until it reaches the orientation associated to **R**
- The Euler theorem guarantees that we can reach any orientation in this way and there are many alternatives to decide the axis of rotations, the only constraint being that two consecutive rotations occur along orthogonal directions
- x-y-z,x-z-y,y-z-x,y-x-z, **z-x-y**, z-y-x (**Tait-Brayn angles, yaw-pitch-roll**)
- x-y-x, x-z-x , y-z-y, x-z-x ,z-x-z, z-y-z (**Euler angles**)
- Rotation direction can be along the mobile frame DF (**intrinsic rotations**) or the fixed frame ENU (**extrinsic rotations**) so we have 24 possibilities to reach the final orientation

Orientation Composition

- Let assume to have two devices A and B, one with orientation ${}^{ENU}\mathbf{R}_B$ and ${}^{ENU}\mathbf{R}_A$
- If ${}^{A}\mathbf{R}_B$ is the orientation of device B respect to A, i.e., when assuming A be a 'fixed' frame,
- then ${}^{ENU}\mathbf{R}_B = {}^{ENU}\mathbf{R}_A \, {}^{A}\mathbf{R}_B$

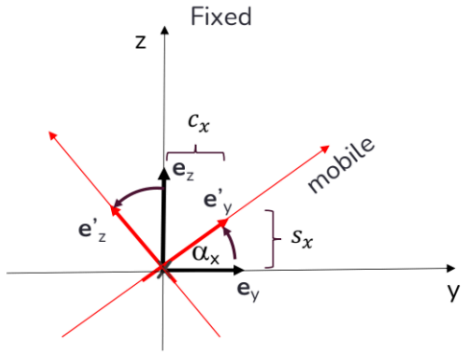
$${}^{ENU}\mathbf{R}_B = {}^{ENU}\mathbf{R}_A \, {}^{A}\mathbf{m} = {}^{ENU}\mathbf{R}_A ({}^{A}\mathbf{R}_B \, {}^{B}\mathbf{m}) = ({}^{ENU}\mathbf{R}_A \, {}^{A}\mathbf{R}_B) \, {}^{B}\mathbf{m}$$

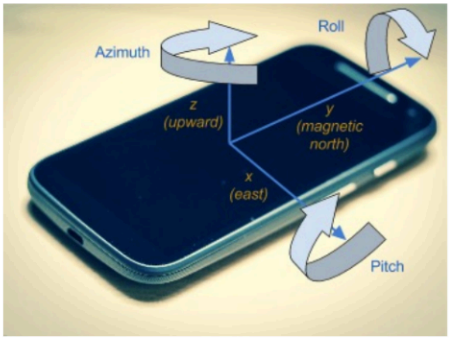
$${}^{A}\mathbf{m} = {}^{A}\mathbf{R}_B \, {}^{B}\mathbf{m}$$

- $\mathbf{R}_z$ → Yaw (Azimuth): degrees of rotation about the z axis
- $\mathbf{R}_x$→ Pitch: degrees of rotation about the x axis
- $\mathbf{R}_y$ → Roll: degrees of rotation about the y axis

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_x & s_x \\ 0 & -s_x & c_x \end{bmatrix}$$

$c_x = \cos(\alpha_x)$
$s_x = \sin(\alpha_x)$





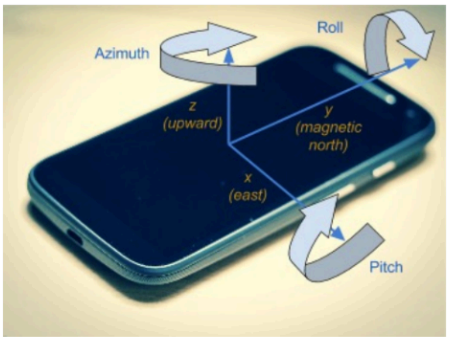
- $\mathbf{R}_z$ → Yaw (Azimuth): degrees of rotation about the z axis
- $\mathbf{R}_x$→ Pitch: degrees of rotation about the x axis
- $\mathbf{R}_y$ → Roll: degrees of rotation about the y axis

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_x & s_x \\ 0 & -s_x & c_x \end{bmatrix}$$

$$\mathbf{R}_y = \begin{bmatrix} c_y & 0 & s_y \\ 0 & 1 & 0 \\ -s_y & 0 & c_y \end{bmatrix}$$

$$\mathbf{R}_z = \begin{bmatrix} c_z & s_z & 0 \\ -s_z & c_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \mathbf{R}_z\mathbf{R}_x\mathbf{R}_y$$



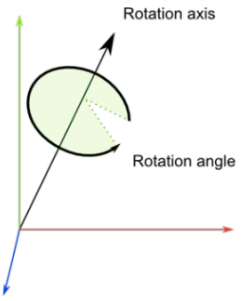
HOW TO RETRIVE ORIENTATION ANGLES FROM R?

$$\mathbf{R} = \mathbf{R}_z\,\mathbf{R}_x\,\mathbf{R}_Y = \begin{bmatrix} c_z & s_z & 0 \\ -s_z & c_z & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & c_x & s_x \\ 0 & -s_x & c_x \end{bmatrix}\begin{bmatrix} c_y & 0 & s_y \\ 0 & 1 & 0 \\ -s_y & 0 & c_y \end{bmatrix} = \begin{bmatrix} * & c_x s_z & * \\ * & c_x c_z & * \\ -c_x s_y & -s_x & c_x c_y \end{bmatrix}$$

$$azimuth = tg^{-1}\left(\frac{r_{12}}{r_{22}}\right)$$
$$pitch = sin^{-1}(-r_{32})$$
$$roll = tg^{-1}\left(\frac{-r_{31}}{r_{33}}\right)$$

Quaternions

- The quaternions (those with unit length) basically encode the coordinates of the rotation axis and the angle in a 'nicer' way
- Formally, quaternions are an extension of the complex numbers, likewise complex numbers are an extension of real numbers
- Quaternions are **pairs of complex numbers** (and 'trions' cannot exit)
  - Unit length quaternions are rotations in 3D likewise unit length complex numbers are rotations in 2D (remember from your study in math?)
- Hence a quaternion is: $\dot{\mathbf{q}} = \mathbf{c}_1 + \mathbf{j}\mathbf{c}_2$, $\mathbf{c}_1 = (a_1 + \mathbf{i}b_1)$, $\mathbf{c}_2 = (a_2 + \mathbf{i}b_2)$, where **j** is a new imaginary unit
- How to define the product among **i** and **j** ?

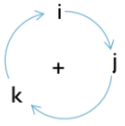


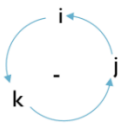
QUATERNION MULTIPLICATION

- The mathematician Hamilton 'discovered' that the product among **i** and **j** is another imaginary unit **k**
- **ij=k**, **jk=i**,**ki=j**
- And that the multiplication is not commutative (revolutionary idea for that time) :
- **ji=-k**, **kj=-i**,**ik=-j**
- Moreover, $\mathbf{i}^2=\mathbf{j}^2=\mathbf{k}^2=-1$
- With this definitions $\mathbf{c}_1+\mathbf{j}\mathbf{c}_2 = a+\mathbf{i}b+\mathbf{j}c+\mathbf{k}d = \dot{\mathbf{q}}$
- For b=c=d=0, $\dot{\mathbf{q}}$ is as a real number...
- For c=d=0, $\dot{\mathbf{q}}$ is a complex number..
- ... and for a=0, $\dot{\mathbf{q}}$ embeds a 3D vector!

×	1	**i**	**j**	**k**
1	1	**i**	**j**	**k**
i	**i**	−1	**k**	−**j**
j	**j**	−**k**	−1	**i**
k	**k**	**j**	−**i**	−1

«Here as he walked by
on the 16th of October 1843
Sir William Rowan Hamilton
in a flash of genius discovered
the fundamental formula for
quaternion multiplication
$i^2 = j^2 = k^2 = i\,j\,k = -1$
and cut it on a stone of this bridge.»





WHY QUATERNIONS?

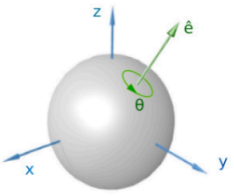
- There are several advantages when using quaternions to represent orientations:
  - Operations are more efficient than the rotation matrix counterpart
  - Being 'numbers' they enjoy amenities like interpolations of two quaternions, like y=lx+(1-l)x
  - No singularity for 'pathological' orientations (Gimbal lock)
- Aside: Quaternions are used in some ANN instead of real numbers and proof to be more efficient

QUATERNION LENGHT

- Beside pair of complex numbers, a quaternion there are two additional representations
- the sum of a scalar s and a 3D vector **v**, $\dot{\mathbf{q}} = s + \mathbf{v}$ ("scalar plus vector" notation)
- a 4D vector $\dot{\mathbf{q}}$ =(a,b,c,d) , a=s, b=v_1,c=v_2,d=v_3
- The *length* of $\dot{\mathbf{q}}$ is defined as $|\dot{\mathbf{q}}| = \sqrt{s^2 + v_1^2 + v_2^2 + v_3^2}$

HOW UNIT QUATERNIONS ENCODE ROTATIONS

- A rotation of θ around axis **e** is associated to the quaternion $\dot{\mathbf{q}} = \cos \theta/2 + \mathbf{e} \sin \theta/2$
- **e** =(x,y,z) is the rotation axis, |**e**|=1
- A unit quaternion $\dot{\mathbf{q}} = s+\mathbf{v}$ represents the rotation of $\theta=2\cos^{-1}(s)$, around $\mathbf{r} = \frac{1}{\sqrt{1-s^2}}\mathbf{v}$



EXAMPLES

Quaternion	Axis-angle	Description
(1,0,0,0)	(undefined,0)	Identity
(0,1,0,0)	(1,0,0),180	Rotation around x axis (pitch)
(0,0,1,0)	(0,1,0),180	Rotation around z axis (yaw)
(0,0,0,1)	(0,0,1),180	Rotation around y axis (roll)
$(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0, 0)$	(1,0,0), 90	Rotation around a axis (pitch)

$$\mathbf{q}_{roll} = [\cos(\theta/2),\ \sin(\theta/2)[0,\ 0,\ 1]]$$
$$\mathbf{q}_{pitch} = [\cos(\theta/2),\ \sin(\theta/2)[1,\ 0,\ 0]]$$
$$\mathbf{q}_{yaw} = [\cos(\theta/2),\ \sin(\theta/2)[0,\ 1,\ 0]]$$

QUATERNION MULTIPLICATION

- Given two rotations, $\mathbf{R}_1$ and $\mathbf{R}_2$, the rotation $\mathbf{R}=\mathbf{R}_1\mathbf{R}_2$ is represented by the quaternion $\dot{\mathbf{q}}_1 \oplus \dot{\mathbf{q}}_2$ Q, where $\dot{\mathbf{q}}_1$ represents $\mathbf{R}_1$ and $\dot{\mathbf{q}}_2$ $\mathbf{R}_2$
- $\dot{\mathbf{q}}_1 \oplus \dot{\mathbf{q}}_2 = (a_1,b_1,c_1,d_1) \oplus (a_2,b_2,c_2,d_2) = (a_3,b_3,c_3,d_3)$

$$\begin{aligned}
a_3 &= a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 \\
b_3 &= a_1b_2 + a_2b_1 + c_1d_2 - c_2d_1 \\
c_3 &= a_1c_2 + a_2c_1 + b_2d_1 - b_1d_2 \\
d_3 &= a_1d_2 + a_2d_1 + b_1c_2 - b_2c_1.
\end{aligned}$$

$\times$	1	$\mathbf{i}$	$\mathbf{j}$	$\mathbf{k}$
1	1	$\mathbf{i}$	$\mathbf{j}$	$\mathbf{k}$
$\mathbf{i}$	$\mathbf{i}$	-1	$\mathbf{k}$	$-\mathbf{j}$
$\mathbf{j}$	$\mathbf{j}$	$-\mathbf{k}$	-1	$\mathbf{i}$
$\mathbf{k}$	$\mathbf{k}$	$\mathbf{j}$	$-\mathbf{i}$	-1

Computation complexity of quaternion multiplication: 16 multiplications 12 sums = 28
Computation complexity of matrix multiplication: 27 multiplication, 18 sums = 45

Rotation Vector and Game Rotation Vector

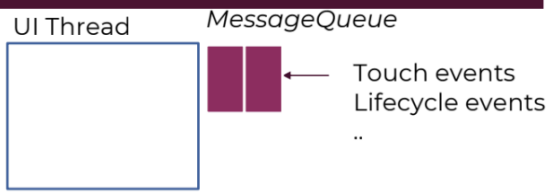
- Compact representation of quaternion as the vector part of a unit quaternion
- The quaternion $\dot{\mathbf{q}}$ =cos $\theta/2$+ **e** sin $\theta/2$, is represented as **v**=sin($\theta/2$)**e,**
- The **rotation vector** is a virtual sensor that returns the vector **v**=**e** sin($\theta/2$)
  - Where **e** =(x,y,z) is the rotation axis, to mean $\dot{\mathbf{q}}$ =cos $\theta/2$+ **e** sin $\theta/2$
  - Since |**e**|=1, the angle $\theta/2$ can be inferred from |**v**|
- The rotation vector is estimated using accelerometer, magnetometer and gyroscope
- The **game rotation vector** sensor is identical to the rotation vector, except it does not use the geomagnetic field

Parallel computing

Threads

MAIN THREAD

- An android app is a multithreading software because this allows to exploit multi-core and to achieve responsiveness
- The UI thread (or **main thread**) is designed as a reactive software, this means it continuously reads from a queue that contains events to process
- Heavy work must run outside the UI thread otherwise the UI becomes unresponsive



time distance among two *onDraw* approx 16 ms

- In reality, threads spend a lot of time waiting for data to be fetched from disk, network, etc.
- The number of threads that can be launched is limited by the underlying operating system (each takes some number of MBs).
- Threads aren't cheap, as they require context switches which are costly.
- Working with threads is hard. Bugs in threads (which are extremely difficult to debug), race conditions, and deadlocks are common problems we suffer from in multi-threaded programming.
- Threads terminating due to exceptions is a problem that deserves to be a separate point.

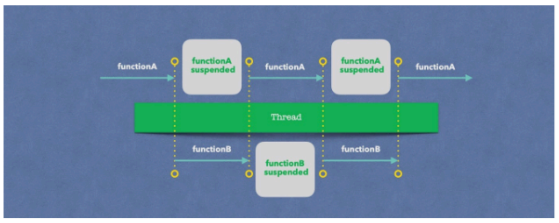
Coroutines

ASYNCRONOUS PROGRAMMING STYLE

- Asynchronous programming aims at avoiding the inefficiency of thread paradigms
- Instead of waiting for the result of a function at the callsite, in async programming one passes what is to be done with the result when it is available. This is the concept of **callback**

- Kotlin has the **suspend** keyword, which marks functions that at some point wait for something (get blocked), meaning they can be moved away from the execution and brought back later. Functions marked with this keyword are called suspending functions or **coroutines**.
- Code with suspending functions looks like ordinary sequential code, but under the hood everything is done asynchronously and effectively (the complier modifies the original structure of the code using a state machine, see documentation for details).

- A *coroutine* is an instance of **suspendable** computation.
- It is conceptually like a thread: it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any specific thread.
- It has suspendable points, so that it may suspend its execution in one thread and resume in another one.
- When a coroutine is suspended, that thread is free for other coroutines.



Scope

The **scope** makes the code lifecycle-aware (it is ensured that the code ends when the calling software component ends, in this case the viewModel – see later)

- The topmost scope is the **GlobalScope**
- Android provides additional scopes, like the **ViewModel** scope meaning that the coroutines launched with this scope end when the ViewModel ends

- The ViewModel class is a business logic or screen level state holder.
- It exposes state to the UI and encapsulates related business logic.
- Its principal advantage is that it caches state and persists it through configuration changes.
- This means that your UI doesn't have to fetch data again when navigating between activities, or following configuration changes, such as when rotating the screen.

- The threads inside which coroutines run are specialized for specific computations
- The threads are identified by a Dispatcher label
- For example, Dispatcher.IO manages a pool of threads created and shutdown on demand
- Code that perform IO operations must be executed inside this Thread pool

```
fun login(username: String, token: String) {
    // Create a new coroutine to move the execution off the UI thread
    viewModelScope.launch(Dispatchers.IO) {
        val jsonBody = "{ username: \"$username\", token: \"$token\"}"
        loginRepository.makeLoginRequest(jsonBody)
    }
}
```

Dispatchers

In the context of coroutines, particularly in Kotlin, dispatchers are a key concept that determines what thread or threads the coroutine code will run on. Dispatchers are part of the Kotlin Coroutine library and are used to control the execution of coroutines, providing a way to specify the thread for coroutine execution.

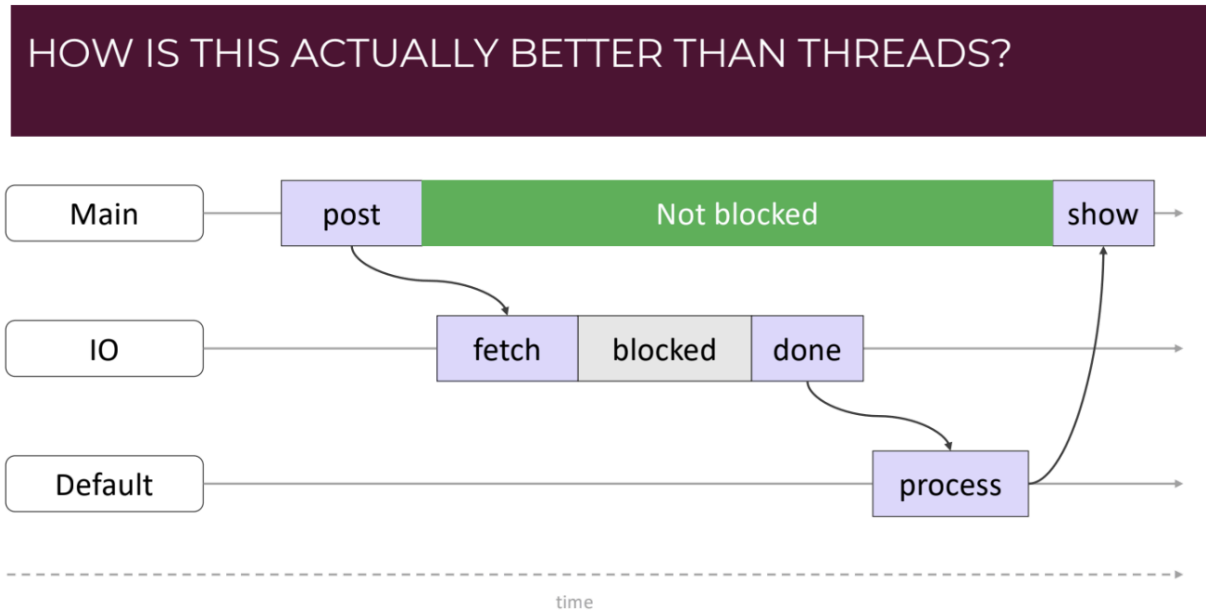
Types of Dispatchers:

- **Default**: This dispatcher is optimized for CPU-intensive work. It uses a shared background pool of threads. The number of threads used depends on the number of CPU cores but is at least two.
- **IO**: This dispatcher is optimized for I/O-related tasks, like reading and writing from the file system, database operations, or network calls. It also uses a shared pool of threads, but the thread count is not limited by the CPU cores, allowing for a higher level of parallelism for I/O operations.
- **Main**: This dispatcher is confined to the main thread of the application (often used in UI applications). It's primarily used for interacting with the UI or performing work that needs to be done on the main thread, like updating UI components.
- **Unconfined**: The Unconfined dispatcher starts a coroutine in the caller thread, but only until the first suspension point. After suspension, the coroutine resumes in the thread of the corresponding suspending function, which might be different. This dispatcher is not confined to a specific thread and can be used for advanced use cases.

Threads vs Coroutines

1. **Basic Concept**:
   - **Threads**: A thread is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically part of the operating system. Threads are a way to achieve multitasking and concurrency, and they run in parallel on separate processor cores.
   - **Coroutines**: Coroutines, on the other hand, are a form of cooperative multitasking where the control flow is explicitly passed between different routines without returning. Coroutines are managed at the application level rather than the operating system, and they don't necessarily run in parallel or on separate processor cores.
2. **Concurrency and Parallelism**:
   - **Threads**: Threads can achieve true parallelism (running on multiple cores simultaneously), but this comes with complexity in synchronization and potential issues like race conditions and deadlocks.
   - **Coroutines**: Coroutines provide concurrency but not necessarily parallelism. They allow different tasks to be interleaved, so while they might not be running at the same instant, they can still progress concurrently in a single-threaded or multi-threaded environment.
3. **Overhead**:
   - **Threads**: Threads are heavier in terms of system resources. Creating and context switching between threads can be resource-intensive because each thread requires its own stack and OS resources.
   - **Coroutines**: Coroutines are lightweight. They require less overhead than threads because they don't need their own stack, and context switching between coroutines is generally less resource-intensive. This allows for creating thousands or even millions of coroutines in a single program.
4. **Control**:
   - **Threads**: Thread execution is managed by the operating system. The OS scheduler decides when to switch between threads, and this switching can occur at any moment.
   - **Coroutines**: Coroutines provide more control over their execution. The coroutine decides when to yield execution to another coroutine, enabling cooperative multitasking.

5. **Use Cases**:
    - **Threads**: Threads are suitable for CPU-bound tasks and scenarios where true parallelism is beneficial, like processing large data sets, performing complex calculations, or handling multiple I/O operations in parallel.
    - **Coroutines**: Coroutines are ideal for I/O-bound tasks and asynchronous operations, such as network calls, database operations, or UI interactions where you want to avoid blocking the main thread.
6. **Synchronization**:
    - **Threads**: Synchronization in threading is critical and often complex, involving mechanisms like locks, semaphores, or concurrent data structures to manage shared resources.
    - **Coroutines**: Coroutines often simplify synchronization because they run in a single thread by default, reducing the need for complex synchronization primitives.

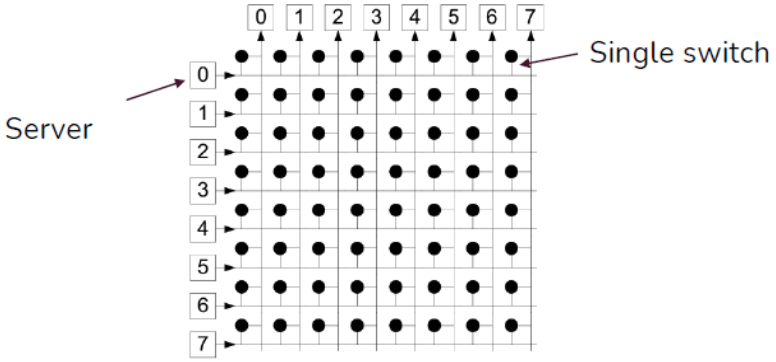


Cloud Computing

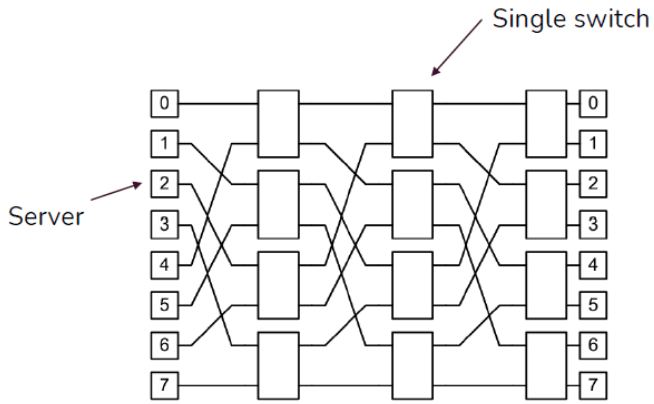
Cloud computing is a **utility** providing computing service; hardware and software resources are concentrated in large data centers, and users buy a computing service, paying for what they consume. Virtualization is the key to sell a slice of a server.

These large data centers are not only a collection of co-located servers wired up together; indeed, big services like Gmail run on **Warehouse-Scale Computers (WSC)**, exploiting clusters of hundreds of thousands of individual servers. All the connected servers are seen as a very big computer.

The servers in a WSC are connected hierarchically. The single servers are first grouped into cluster switches; then, the cluster switches are connected among themselves.
Let us consider a group of servers connected by the means of a cluster switch. In order to ensure the existence of a communication path between each pair of servers at the same time, the cluster switch should have a completely connected topology, thus giving rise to the so-called **crossbar switch**:



However, the cost of this type of switch is very high: if N is the number of servers grouped into the switch, the cost of the crossbar switch is $O(N^2)$.
In order to reduce this high cost, another kind of switch can be employed; we are talking about the **omega network**:



This network is a multi-stage network in which any input can be connected to any output within a single switch. This solution allows to have a path from any server to any other one, assuming that the network has been properly arranged a priori.
Although this solution is more convenient with respect to the previous one (it costs $O(NlogN)$), there could be *collisions* in the communication: some connections cannot take place at the same time.

We can classify the software in three categories:

- **Server-level software**: firmware, operating system and libraries of the single server.
- **Cluster-level software**: it's a kind of operating system of the datacenter; for instance, distributed file systems.
- **Application-level software**: software implementing online (e.g., email), offline (e.g., data analysis) or cloud services.

In general, cloud computing can be defined under two different points of view:

- **Deployment model**: There are four main categories of cloud deployment:
    - Public cloud: computing resources are owned by a third-party cloud service provider; these resources are made accessible through the Internet. Users typically pay for the services on a pay-as-you-go basis.

    - Private cloud: computing resources are used exclusively by a single organization.
    - Community cloud: cloud that is shared by multiple organizations with common interests or requirements.
    - Hybrid cloud: combines elements of both public and private clouds, allowing data and applications to be shared between them. Organizations can use the public cloud for scalable and non-sensitive computing tasks while keeping critical workloads or sensitive data in a private cloud.

- **Service delivery model**: type of computing service provided by a cloud service provider to its users. The three primary categories of service delivery models are the following:
    - Infrastructure as a Service (IaaS): provides a virtualized infrastructure, giving users the control over the underlying operating systems and applications.
    - Platform as a Service (PaaS): offers a platform with development tools, allowing developers to focus on building the final product.
    - Software as a Service (SaaS): delivers fully functional software applications as a service, eliminating the need for users to install, maintain or manage the underlying infrastructure.

There are five different characteristics of a cloud:

- **On-demand self-service**: ability that empowers users to consume the computing facilities as much as they need at any given moment. The user can request by himself/herself cloud services as needed through some interface, and the requested resources become available within seconds.

- **Resource pooling**: computing resources are managed as a pool, which is maintained by the provider at a remote location and is

accessed by all the users. In cloud computing, the concept of *multi-tenancy* is very important: multi-tenancy in fact allows the sharing of the same computing resources by different subscribers, without the subscribers being aware of it.

- **Measured service**: there exists a formal agreement between a provider and a consumer of a service, defining two metrics:
    - Service Level Indicator (SLI): metric that can be monitored (e.g. response time, uptime);
    - Service Level Objective (SLO): a condition on a measure of a specific metric (e.g., the mean response time must be at most of 1s).

  This formal agreement defining both the SLI and the SLO is called **Service Level Agreement (SLA)**; there is a penalty in case of violation of a SLA.

- **Rapid elasticity**: the ability of a cloud service to quickly and easily scale resources up or down based on demand. This elasticity allows users to dynamically allocate and deallocate computing resources, such as virtual machines, storage, and network bandwidth, in response to changing workloads. How can we effectively determine the needed amount of resources? There are two approaches:
    - Traditional solution: Resources are provisioned based on peak usage predictions and expected workloads. Organizations often over-provision to ensure that they have enough capacity to handle maximum demand, but this leads to higher costs.
    - Autonomic solution: the system adapts to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.

- **Broadband access**: cloud resources are accessed over the internet by using standard access mechanisms that provide platform-independent access, such as public APIs.

In what follows, we will briefly describe the differences between cloud computing, cluster computing, distributed computing and grid computing.

- **Cloud computing**: model for delivering and consuming computing resources over the Internet on a pay-as-you-go basis.

- **Cluster computing**: interconnection of multiple computers (nodes) to work together as a single integrated system for solving complex computational problems.

- **Distributed computing**: use of multiple interconnected computers that work together in order to achieve a common goal.

- **Grid computing**: coordinated use of a collection of diverse and geographically dispersed resources in order to solve large-scale computational problems.

Problem: some applications, like for instance self-driving cars, use large data volumes originated from mobile and IoT devices, and they may need low latency. Cloud-only solutions may be impractical due to these low latency requirements!
→ We could move the computation and the storage capabilities closer to the end user: **Fog Computing**!

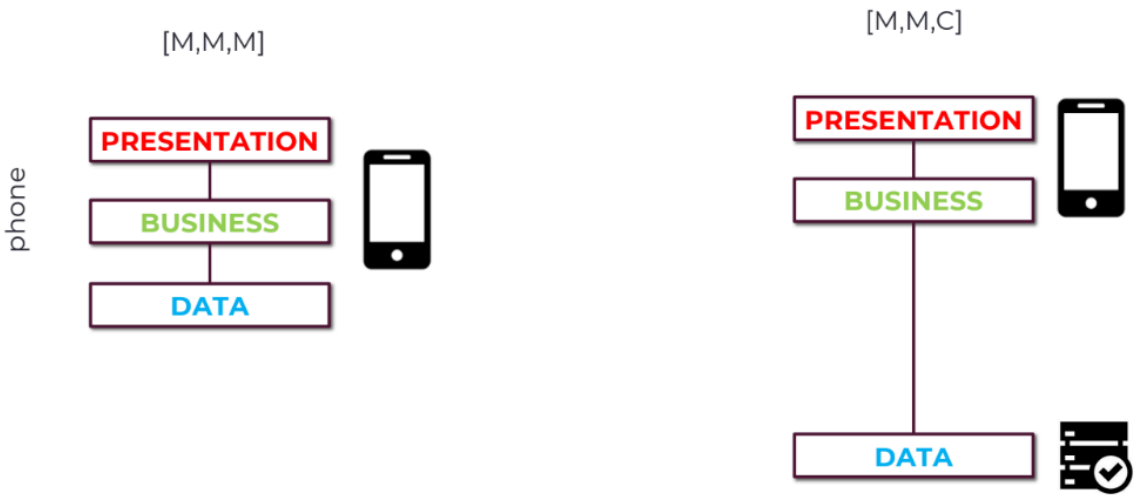
Fog computing is a distributed computing paradigm that extends cloud computing capabilities to the *edge of the network*, closer to the data source or devices that produce and consume data.

The *edge* of a network is the outer boundary where the network interfaces with the end user device and other external systems. It is the point at which data is generated, consumed, or enters/leaves the network. In networking, there are two kinds of edges:

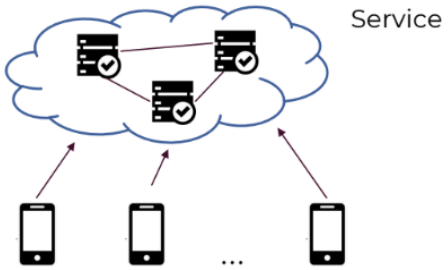
- **User edge**: it is the point at which data is generated and enters the network.
- **Device-to-Device edge**: it is the point at which data processing occurs close to the source, thus reducing latency and optimizing resource usage.

Overall, fog computing complements traditional cloud computing by distributing computing resources strategically across the network, enabling more efficient and responsive systems.

The fog platform provides low-latency virtualized services, and it is linked to the cloud computing infrastructure. Edge devices request computation, storage and communication services from the fog platform; the fog platform provides local, low-latency responses to these requests, and forwards relevant data for computationally intensive processing.



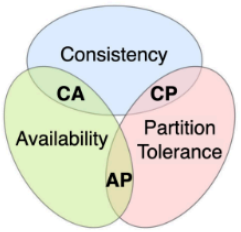
- When splitting the app in two parts, the remote site holds some useful data (stateful)
- We can think at the remote site as providing a stateful service (more on this later)
- To increase the capacity to serve clients (throughput) , the service is usually replicated on several servers



CAP Theorem

- **C**onsistency[informally]: Once a client updates the state, then all the other clients see the same latest value of the state or an error
  - Any write is immediately replicated on the other site before any read is processed
  - The service can reply with an error message (e.g., data not available) due to sync failures with the other site
- **A**vailability[informally]: Every read receives a meaningful response, i.e. no error, that may be not the latest one
- **CA** can be ensured if there are no partitions between the two sites
- **P**artition tolerance[informally]: The service continues to operate, despite an arbitrary number of messages exchanged among the replicas are dropped
- **AP**: Each site replies with the latest update (can be different on the other site)
- **CP**: A site can reply with an unviability error , but when replying with data, they are consistent

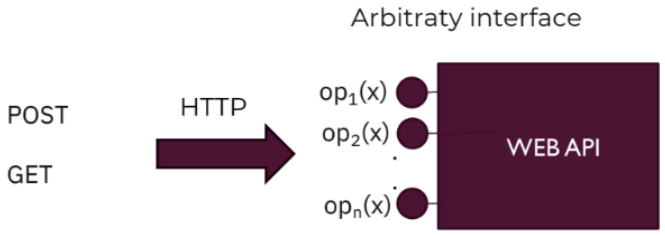
- Claim: Any distributed stateful service can provide any two of the three **CAP** properties
- Any distributed stateful service cannot provide all the three **CAP** guarantees
- For example, if the service works even under portioning, then either the service guarantees only consistency (**CP**), i.e. sometimes it is unavailable, or availability, i.e. It always responds but sometimes the replies are outdated (**AP**).
- Partition tolerance usually applies to noSQL DB (like firebase)



Rest Model for remote services

- The remote service is modeled as a **Resource**
- With a fixed (not expandable) set of **self-contained** and **independent** operations (operations can't refer to previous operations)
- Philosophy: An operation triggers a state transition of the resource. A representation of the new *state* is *transferred* back to the client, so that the client has all it needs to proceed
- REST = **RE**presentational **S**tate **T**ransfer
- **RESTful** service: A service that adheres to these design rules
- RESTful web service: restful service that uses web protocols, like http, or https

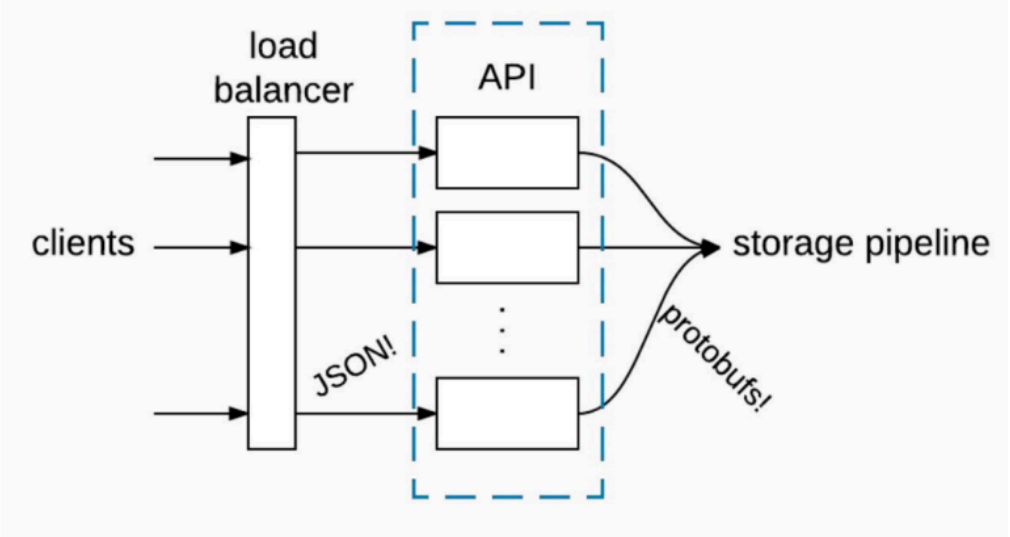
Web API



- A web API is a service that is reachable via HTTP and provides an arbitrary interface to clients
- Data are usually exchanged in the JSON format, or even XML
- RESTful web API means a restful service that uses http or https and JSON/XML

Resource Management for Cloud Services

- When design a cloud service it is important to address non-functional requirements, like
- How long does a request wait before being served?
- What percentage of requests wait at least t ms before being processed?
- How many server replicas are needed to ensure that the average service time is less than t?
- What is the availability of the service, i.e. the percentage of time a client request is served?



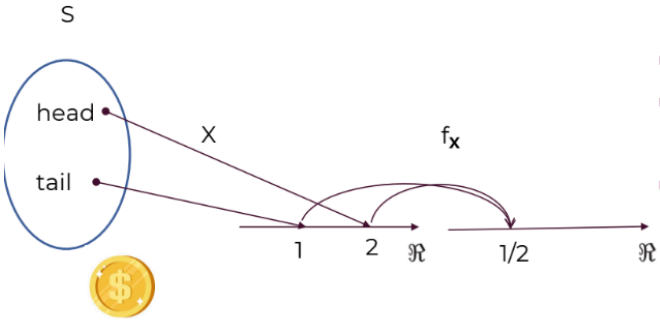
1.Question: How do we allocate appropriate resources for this service?

- There are four main approaches for the implementation of resource management policies are based on

A. Control theory
B. Utility
C. Machine learning
D. Optimization theory

RECAP ON RANDOM VARIABLES (RV)

Definition (Random Variable). A random variable X on a sample space S is a function $X : S \to \Re$ that assigns a real number $X(s)$ to each sample point $s \in S$.



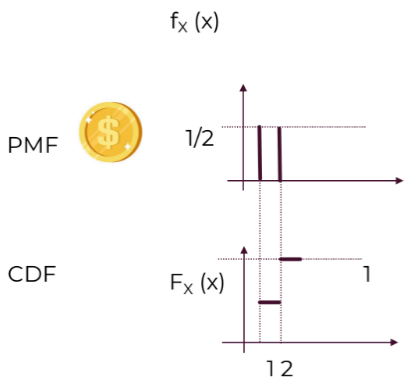
- If the image of X is finite then X is a discrete random variable, otherwise it is a continuous random variable
- If X is discrete, then X is characterized by
- PMF (probability mass function)

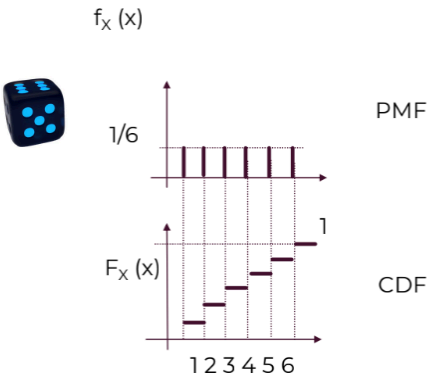
$$f_X(x) = \Pr\{X=x\}$$

- CDF (Cumulative Distribution Function)

$$F_X(x) = \sum_{t \le x} f_X(t)$$

EXAMPLE 1





Discrete-Time Markov Chains (DTMC)

- A Discrete-Time (homogeneous) Markov chain is defined by

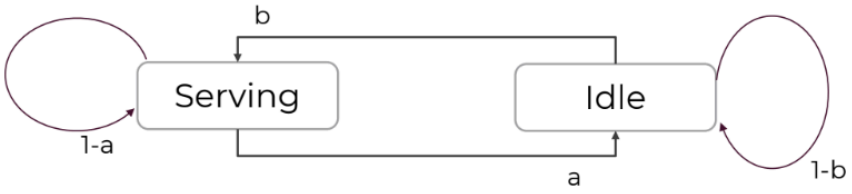
1. Finite set of states S
2. Transition probability function p:SxS→[0,1],
   - 1-step transition probability $Pr\{X_{n+1}=j|X_n=i\}=p_{ij}(1)=p_{ij}$
   - Assigned in the form of a stochastic matrix (matrix with each row summing to 1)
3. initial state distribution of X_0

Markov Reward Model (MRM)

- MRM extends a DTMC (or CTMS see later) by adding a reward rate to each state.
- Reward : r:S→R
- At the steady state one can measure the average reward $E[r]=\Sigma\, v_i r_i$
- An additional variable can record the reward accumulated up to the current time, but this setting needs more elaboration as the accumulated reward can be infinity.

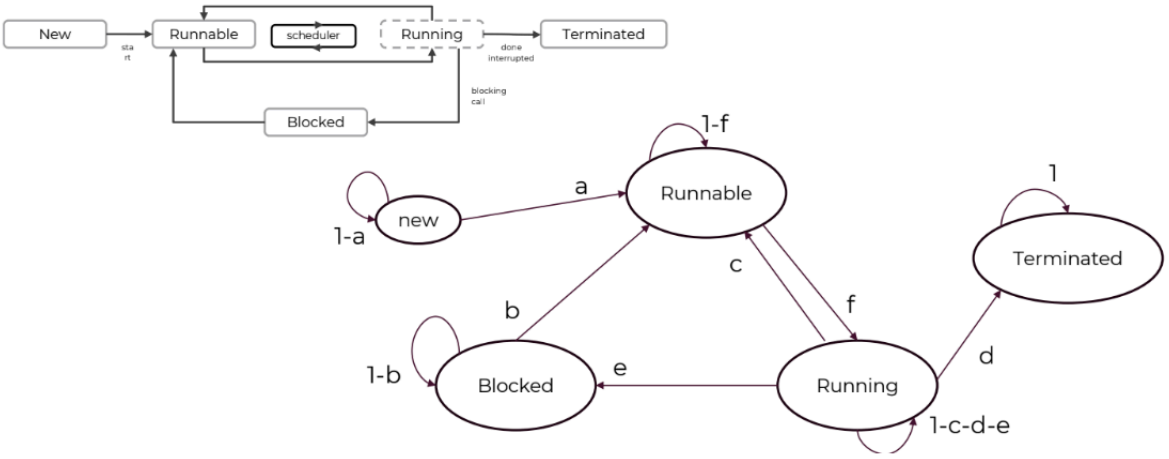
EXAMPLE 1: SERVER

- System: server. The state of the server at each time tick can either be serving or idle
- We can model the dynamic of the server as a memoryless system by assuming (0<a,b<1)
  - We don't know if this assumption is true..
- $Pr\{X_{n+1}=idle|X_n=serving\}=a$, $Pr\{X_{n+1}=serving|X_n=Idle\}=1-a$
- $Pr\{X_{n+1}=serveing|X_n=idle\}=b$, $Pr\{X_{n+1}=idle|X_n=serving\}=1-b$



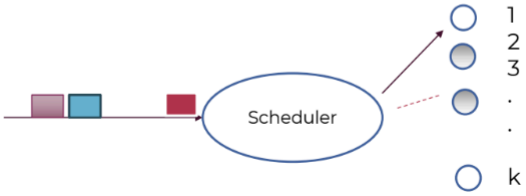
EXAMPLE 2: THREAD MODEL

- The state of a thread can be S={RUNNABLE,BLOCKED,RUNNING,TERMINATED}



EXAMPLE 3: RR SCHEDULER

- A server with k cores receives requests from different clients via the same physical connection
- At regular intervals, a scheduler may decide to accept a new service request, and to reply to a client if its service ends

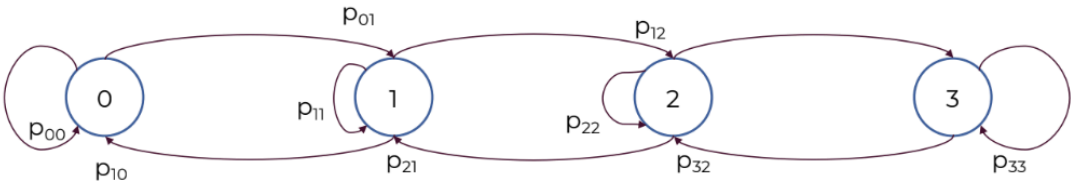


1

EXAMPLE 3: RR SCHEDULER WITH K=3

- S={"request arrive", 'request doesn't arrive", "service ends', 'service continues'}
- S={A,A'E,E'}
- Events = {A,AE,AE',A'E,A'E'}

- p_{00}=Pr{A'}, p_{01}=Pr{A}
- p_{10}=Pr{A'}Pr{E}, p_{11}=Pr{A}Pr{E}+Pr{A'}Pr{E'}, p_{12}=Pr{A} Pr{E'}
- p_{21}=Pr{A'}Pr{E}, p_{22}=Pr{A}Pr{E}+Pr{A'}Pr{E'}, p_{23}=Pr{A} Pr{E'}
- p_{32}=Pr{A'}Pr{E}, p_{33}=Pr{A}Pr{E}+Pr{A'}Pr{E'}+Pr{A}Pr{E'}



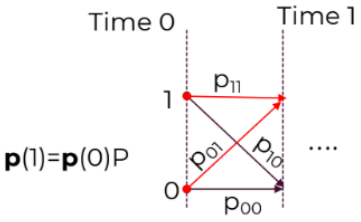
EVOLUTION OF THE MC

- How to find the distributions X_n for any n?
- Because the system is memoryless, we can easily find the distribution of X_{n+1}, how?
- Distribution of X_{n+1} means to determine the probability of the next possible states
- By applying the total probability law...
- $Pr\{X_{n+1}=j\}=\sum_i Pr\{X_n=i\}\ p_{ij}$

[1] RR stands for Round robin

EXAMPLE

- Set of states: [0,1]
- $P=\begin{bmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{bmatrix}$
- Initial State probability: $\mathbf{p}(0)=[p_0(0),p_1(0)]$

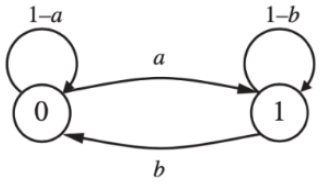


$\mathbf{p}(1)=\mathbf{p}(0)P$

- State probability at time 1:
- $Pr\{X_1=0\}=Pr\{X_0=0\}p_{00}+ Pr\{X_0=1\}\ p_{10}$
- $Pr\{X_1=1\} =Pr\{X_0=1\}\ \ p_{11} + Pr\{X_0=0\}p_{01}$
- $\mathbf{p}(1)= [p_0(1) \quad p_1(1)]$
- $[p_0(1) \quad p_1(1)] =[p_0(0) \quad p_1(0)] \begin{bmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{bmatrix}$

by iterating, $\mathbf{p}(2)=\mathbf{p}(1)P=\mathbf{p}(0)PP=\mathbf{p}(0)P^2$...

$\mathbf{p}(n)= \mathbf{p}(0)P^n$



$$P = \begin{bmatrix} 1-a & a \\ b & 1-b \end{bmatrix}, \ 0 \le a, \ b \le 1, \ |1-a-b| < 1,$$

$$P(n) = \begin{bmatrix} \frac{b+a(1-a-b)^n}{a+b} & \frac{a-a(1-a-b)^n}{a+b} \\ \frac{b-b(1-a-b)^n}{a+b} & \frac{a+b(1-a-b)^n}{a+b} \end{bmatrix}.$$

$P^\infty= \frac{1}{a+b}\begin{bmatrix} b & a \\ b & a \end{bmatrix}$

- $p_0(\infty)= \frac{1}{a+b}(p_0(0)b+p_1(0)b)= \frac{b}{a+b}$
- $p_1(\infty)= \frac{a}{a+b}$
- In the limit $n\to\infty$, all rows are the same.
- For large n, $\mathbf{p}(n)$ becomes independent of the initial state (long run behaviour)
- $p_i(\infty)$ is called the limiting state probabilities

A stationary probability vector v is a vector of probabilities such that if the system is in state i with probability v_i, then the probability of being in state j after one step is given by the entry v_j of the vector v. In other words, the stationary probability vector v is a fixed point of the transition matrix P of the DTMC, which means that v is a solution to the equation v = vP. The **stationary probability vector** v is also known as the **steady-state distribution** of the DTMC, and it represents the long-term behavior of the system. Specifically, if the system starts in the stationary distribution, it will remain in this distribution over time.

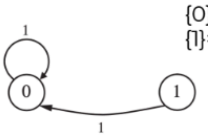
STATIONARITY

- A stationary probability is a vector **v** such $\Sigma v_i=1$ and that $\mathbf{v}=\mathbf{v}P$
- Theorem: If the limiting state probability exits, then $p_i(\infty)=v_i$
- Warning: the converse is not true

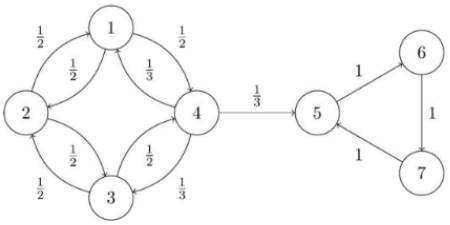
2. **Stationary Distribution**: For a Markov chain, a stationary distribution is a probability distribution over its states such that, if the chain starts in this distribution, the probability distribution of its states in the future (after any number of steps) remains the same. In other words, once the chain reaches a distribution that is stationary, it stays there.

State classification and Limiting Probabilities

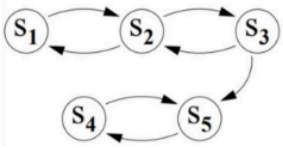
- How to be sure about the long-run beauvoir of a MC?
- States are classified analysing their stochastic properties for n→∞
- Informally
- Transient state : reached a finite number of times and then never return
- Absorbing state: once entered the state cannot change anymore
- Recurrent state : starting from the state, the system returns to that state (with prob. 1)
  - Periodic: the system returns in the state at regular intervals
  - Aperiodic: not periodic



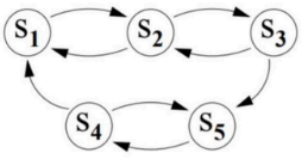
{0}=absorbing
{1}=transient


1
2
3
4
5
6
7
1/2
1/3
1
{5,6,7}=periodic
{1,2,3,4}=transient

- Irreducible MC: every state can be reached from any other state



Not irreducible



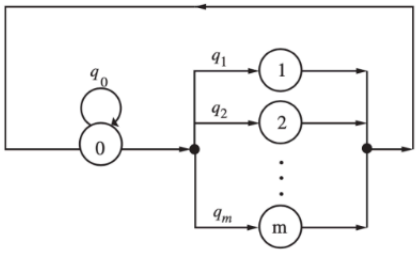
Irreducible

- Theorem: in an irreducible MC all states are of the same type
- Theorem: For any finite, irreducible and aperiodic MC, P(∞) exists
- The steady state can be computed

1. As limits (in the limit n→∞, all rows are the same)
2. Solving the following linear system $\mathbf{v}=\mathbf{v}P$ (replacing one equation with $\Sigma v_i=1$)
3. Power method: repeat until fixed-point is reached: $\mathbf{v}^0$=any, $\mathbf{v}^{k+1}=\mathbf{v}^kP$

EXAMPLE

- Consider a model of a program executing on a computer system with m I/O devices and a CPU.
- The program will be in one of the m+1 states denoted by 0,1,...,m, so that in state 0 the program is executing on the CPU, and in state i(1 ≤ i ≤ m) the program is performing an I/O operation on device i.
- Assume that the request for device i occurs at the end of a CPU burst with probability qi, independent of the history of the program.
- The program will finish execution at the end of a CPU burst with probability q0.
- We assume that the system is saturated so that on completion of one program, another statistically identical program will enter the system instantaneously.



$$P = \begin{bmatrix} q_0 & q_1 & \cdot & \cdot & \cdot & q_m \\ 1 & 0 & \cdot & \cdot & \cdot & 0 \\ \vdots & \vdots & & & \vdots & \\ 1 & 0 & \cdot & \cdot & \cdot & 0 \end{bmatrix} \longrightarrow 1$$

$$v_0 = \frac{1}{2 - q_0}$$

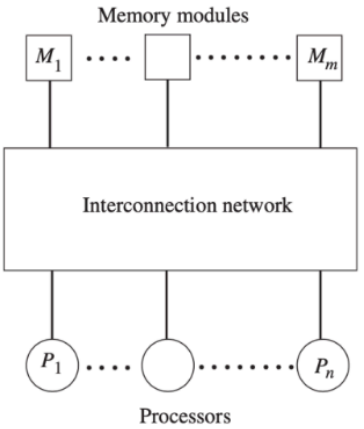
$$\mathbf{v} = \mathbf{v}P \longrightarrow \begin{cases} v_0 = v_0 q_0 + \sum_{j=1}^{m} v_j, \\ v_j = v_0 q_j, \end{cases} \qquad v_j = \frac{q_j}{2 - q_0}$$

the sum of all v is 1

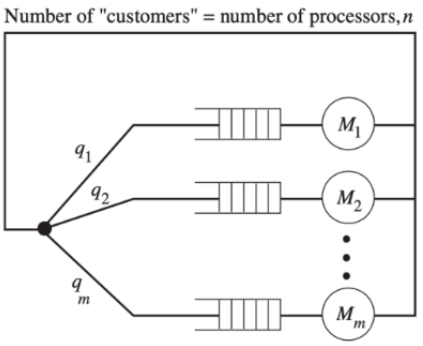
$$\sum_{j=0}^{m} v_j = 1 \longrightarrow v_0 + v_0 \sum_{j=1}^{m} q_j = 1 \longrightarrow v_0(1 + 1 - q_0) = 1$$

EXAMPLE

- Consider the shared memory multiprocessor system shown in Figure. To reduce contention among processors, the memory is usually split up into modules, which can be accessed independently and concurrently.
- When more than one processor attempts to access the same module, only one processor can be granted access, while other processors must await their turn in a queue.
- The effect of such contention, or interference, is to increase the average memory access time.
- Can we compute the average number of memory requests completed per memory cycle for any m,n?

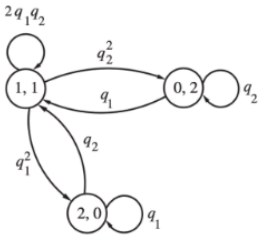


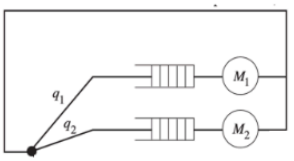
- Assume that the time to complete a memory access is a constant and that all modules are synchronized. Processors are assumed to be fast enough to generate a new request as soon as their current request is satisfied.
- A processor cannot generate a new request when it is waiting for the current request to be completed.
- The operation of the system can be visualized as a discrete-time queuing network as shown in Figure.



- N_1 = number processors waiting or being served by memory module 1
- N_2 = number processors waiting or being served by memory module 2
- State =(N_1,N_2), State space = {(1,1),(2,0),(0,2)}

$$P = \begin{matrix} \\ (1,1) \\ (0,2) \\ (2,0) \end{matrix} \begin{matrix} \begin{matrix} (1,1) & (0,2) & (2,0) \end{matrix} \\ \begin{bmatrix} 2q_1q_2 & q_2^2 & q_1^2 \\ q_1 & q_2 & 0 \\ q_2 & 0 & q_1 \end{bmatrix} \end{matrix}.$$

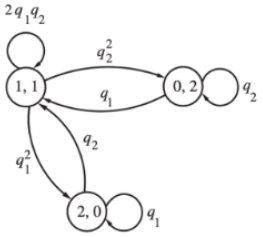




- The solution of the usual linear system

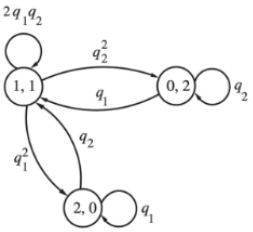
$$\begin{aligned} v_{(1,1)} &= 2q_1q_2v_{(1,1)} + q_1v_{(0,2)} + q_2v_{(2,0)}, \\ v_{(0,2)} &= q_2^2v_{(1,1)} + q_2v_{(0,2)}, \\ v_{(2,0)} &= q_1^2v_{(1,1)} + q_1v_{(2,0)}, \\ v_{(1,1)} &+ v_{(0,2)} + v_{(2,0)} = 1 \end{aligned}$$

$$P = \begin{matrix} \\ (1,1) \\ (0,2) \\ (2,0) \end{matrix} \begin{matrix} \begin{matrix} (1,1) & (0,2) & (2,0) \end{matrix} \\ \begin{bmatrix} 2q_1q_2 & q_2^2 & q_1^2 \\ q_1 & q_2 & 0 \\ q_2 & 0 & q_1 \end{bmatrix} \end{matrix}.$$



$$v_{(2,0)} = \frac{q_1^2}{1-q_1}v_{(1,1)}, \quad v_{(0,2)} = \frac{q_2^2}{1-q_2}v_{(1,1)}, \qquad v_{(1,1)} = \frac{1}{1+\frac{q_1^2}{1-q_1}+\frac{q_2^2}{1-q_2}} = \frac{q_1q_2}{1-2q_1q_2}.$$

- We assign rewards to the three states of the DTMC as follows:
- r(1,1) = 2, r(2,0) = 1, and r(0,2) = 1.
- This rewards represent the number of memory requests completed per memory cycle
- The reward is a rv, with average



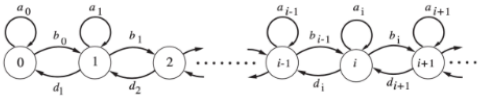
$$\begin{aligned} \mathrm{E}[\mathbf{r}] &= 2v_{(1,1)} + v_{(0,2)} + v_{(2,0)} \\ &= \left(2 + \frac{q_1^2}{1-q_1} + \frac{q_2^2}{1-q_2}\right) v_{(1,1)} \\ &= \frac{1-q_1q_2}{1-2q_1q_2}. \end{aligned}$$

Which is maximum when $q_1=q_2=1/2$, max E[r]=3/2

Birth-Death Markov Chains

- A special type of discrete-time Markov chain with all one-step transitions to nearest neighbours only.
- The transition probability matrix P is a tridiagonal matrix.
- To simplify notation, we let

$$\begin{aligned} b_i &= p_{i,i+1}, \quad i \geq 0 \quad \{\text{the probability of a birth in state } i\}, \\ d_i &= p_{i,i-1}, \quad i \geq 1 \quad \{\text{the probability of a death in state } i\}, \\ a_i &= p_{i,i}, \quad i \geq 0. \quad \{\text{the probability being in state } i\}. \end{aligned}$$



$$P = \begin{bmatrix} a_0 & b_0 & 0 & \cdot & \cdot & \cdot & & & & \\ d_1 & a_1 & b_1 & 0 & \cdot & \cdot & & & & \cdots \\ 0 & d_2 & a_2 & b_2 & 0 & \cdot & & & & \cdots \\ 0 & 0 & \cdot & \cdot & \cdot & \cdot & & & & \cdots \\ \vdots & \vdots & \vdots & \vdots & & & & & & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{i-1} & b_{i-1} & 0 & \cdot & \cdot \cdots \\ 0 & 0 & 0 & 0 & \cdots & d_i & a_i & b_i & 0 & \cdot \cdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & d_{i+1} & a_{i+1} & b_{i+1} & 0 \cdots \\ \vdots & \vdots & \vdots & \vdots & & & & & & \vdots \end{bmatrix}$$

1. **Basic Concept**:
   - A birth-death Markov chain is a continuous-time Markov chain where transitions can occur only between neighboring states.
   - The term 'birth' refers to a transition that increases the state by one, and 'death' refers to a transition that decreases the state by one.
   - These transitions represent the arrival of a new entity into the system (birth) or the departure of an entity from the system (death).
2. **State Space**:
   - The state space of a birth-death Markov chain is typically the set of non-negative integers, where each state represents the number of entities in the system.
   - Transitions can only occur to adjacent states (i.e., from state i to $i+1$ or $i-1$).

3. **Transition Rates**:
   - Each state i has an associated birth rate (λ_i) and a death rate (μ_i).
   - These rates determine the probability per unit time of a birth or a death occurring when the system is in state i.

SOLUTION OF BIRTH-DEATH MC

- **v**=**v**P
- $v_0=a_0v_0+d_1v_1 \rightarrow v_0=(1-b_0)v_0+d_1v_1 \rightarrow 0=-b_0v_0+d_1v_1 \rightarrow v_1=v_0\, b_0/d_1$
- $v_1=b_0v_0+a_1v_1+d_2v_2 \rightarrow v_1= b_0v_0+(1-d_1-b_0)v_1+d_2v_2 \rightarrow v_2=v_1\, b_1/d_2$
- ..
- $v_i=b_{i-1}v_{i-1}+a_iv_i+d_{i+1}v_{i+1} \rightarrow v_{i+1}=v_i\, b_i/b_{i+1}$
- ...

$$v_i = \frac{b_{i-1}}{d_i} v_{i-1} = \prod_{j=1}^{i} \frac{b_{j-1}}{d_j} v_0.$$

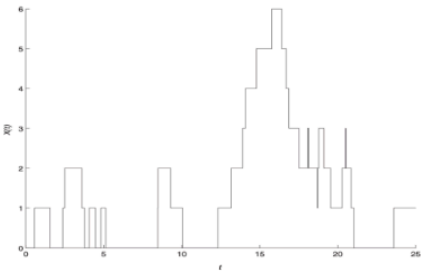
$$v_0 = \frac{1}{\sum_{i\geq 0} \prod_{j=1}^{i} \frac{b_{j-1}}{d_j}}.$$

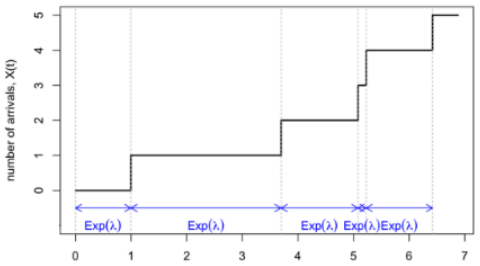
$$P = \begin{bmatrix} a_0 & b_0 & 0 & \cdot & \cdot & \cdot & & & & \\ d_1 & a_1 & b_1 & 0 & \cdot & \cdot & & & & \cdots \\ 0 & d_2 & a_2 & b_2 & 0 & \cdot & & & & \cdots \\ 0 & 0 & \cdot & \cdot & \cdot & \cdot & & & & \cdots \\ \vdots & \vdots & \vdots & \vdots & & & & & & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{i-1} & b_{i-1} & 0 & \cdot & \cdots \\ 0 & 0 & 0 & 0 & \cdots & d_i & a_i & b_i & 0 & \cdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & d_{i+1} & a_{i+1} & b_{i+1} & 0 \cdots \\ \vdots & \vdots & \vdots & \vdots & & & & & & \vdots \end{bmatrix}$$

$$\sum_{i\geq 0} v_i = 1$$

Continuous time Birth-Death Markov Chains

- Let now consider the continuous version of the birth-death process
- The key difference is that a new job can arrive/departure at any time and these events are memoryless
- Our goal is to find how X(t) changes over time





- We only focus on the stationary probability, i.e., we are interested in evaluating the probability that the state of the system is i when t→∞, denoted as π(i)
- To define a CTMC one need to assign the **transition rate** from state i to state j q_{ij}, representing the rate (frequency) with which the state changes from i to j
- The system stays in i for an exponentially distributed amount of time before jumping to another state j
- So, it stays in i for a time T_i = min{T_{i1},T_{i2},...T_{ik}} (which is also exponentially distributed with parameter $q_i=q_{i1}+q_{i2}$ ...+q_{ik}) and then jumps to some state j
- This allows to write a set of simple equations ....

Steady-State:

In steady-state, the probabilities of being in any given state do not change over time, which means the system has reached equilibrium. This is denoted by $\frac{d\pi_0(t)}{dt} = 0$, where $\pi_0(t)$ is the probability of being in state 0 at time t.

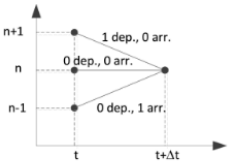
In queuing theory or Markov processes, the notation π_k is often used to denote the probability that the system is in state k in the long run or at steady state. The steady-

STEADY-STATE SOLUTION

- $p_{00}(t,t+dt)=(1-\lambda_0)\ dt$
- $p_{01}(t,t+dt)=\lambda_0\ dt$
- $p_{12}(t,t+dt)=\lambda_1\ dt$
- $p_{11}(t,t+dt)=(1-\lambda_0-\mu_1)\ dt$
- $p_{10}(t,t+dt)=\mu_1\ dt$

$\pi_0(t+dt)=\pi_0(t)\ p_{00}(t,t+dt)+\pi_1(t)\ p_{10}(t,t+dt)$

$\pi_0(t+dt)=\pi_0(t)\ (1-\lambda_0)\ dt+\pi_1(t)\ \mu_1\ dt$



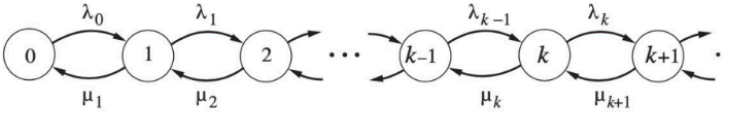
$\pi_0(t+dt)-\pi_0(t) = -\lambda_0\ \pi_0(t)\ dt +\pi_1(t)\ \mu_1\ dt$

$(\pi_0(t+dt)-\pi_0(t))/dt=\pi_0(t)-\lambda_0+\pi_1(t)\ \mu_1$

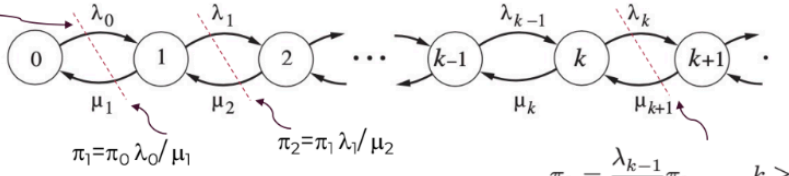
$d\pi_0(t)/dt=\pi_0(t)-\lambda_0+\pi_1(t)\ \mu_1$

At steady-state, the pdf is constant, then $d\pi_0(t)/dt=0$

$\pi_0(t)\lambda_0=\pi_1(t)\ \mu_1 \rightarrow \pi_0\lambda_0=\pi_1\ \mu_1$



balance equation
$\pi_0\lambda_0=\pi_1\ \mu_1$



$$\pi_k = \frac{\lambda_{k-1}}{\mu_k}\pi_{k-1}, \quad k \geq 1.$$

Since $\pi_0+\pi_1+...+\pi_k+..=1 \rightarrow \pi_0(1+\lambda_0/\mu_1+\ \lambda_0/\mu_1\ \lambda_1/\mu_2\ ...+)=1$

$$\pi_k = \pi_0 \prod_{i=0}^{k-1} \left(\frac{\lambda_i}{\mu_{i+1}}\right), \quad k = 1, 2, \ldots, n$$

$$\pi_0 = \frac{1}{1+\sum_{k\geq 1}\prod_{i=0}^{k-1}\left(\frac{\lambda_i}{\mu_{i+1}}\right)}.$$

The balance equation in a Markov process, particularly in the steady-state analysis of a queuing system, is an expression of the equilibrium condition for each state in the system. It ensures that, for each state, the total probability flow into the state equals the total probability flow out of the state. This concept is crucial because it implies that, over time, the distribution of probabilities over the states does not change, which is the definition of steady-state.

The balance equation can be expressed as:

$$\sum_i \pi_i \lambda_i = \sum_j \pi_j \mu_j$$

where:

- π_i is the steady-state probability of being in state i.
- λ_i is the rate at which transitions occur from state i to other states (sometimes referred to as the "birth rate" in a birth-death process).
- μ_j is the rate at which transitions occur into state j from other states (sometimes referred to as the "death rate" in a birth-death process).

In a simple birth-death process like the one shown on your slide, the balance equation for a particular state k can be written as:

$$\pi_{k-1}\lambda_{k-1} = \pi_k \mu_k$$

This equation says that the probability of transitioning out of state $k-1$ into state k (represented by $\pi_{k-1}\lambda_{k-1}$) must be equal to the probability of leaving state k to either go back to state $k-1$ or to move to state $k+1$ (represented by $\pi_k \mu_k$) in the steady-state.

For the entire system, a set of such balance equations can be written for every state, and they can be solved together to find the steady-state probabilities π_k for all states k. The normalization condition mentioned in the slide ensures that the sum of all the steady-state probabilities is 1:

$$\sum_{k=0}^{\infty} \pi_k = 1$$

This is because the probability of being in some state of the system must be certain, and hence the sum of probabilities over all possible states must equal 1. The equations collectively ensure that the system is mathematically balanced and that the probabilities are correctly distributed across the various states of the system.

A special case: MM1

The M/M/1 queue is a basic model in queueing theory that represents the queue length in a system with a single server. The model is characterized by three aspects: the arrival process, the service process, and the number of service channels. Let's break down each component:

1. **Arrival Process (M):** This is represented by a 'Markovian' or 'memoryless' process, typically modeled as a Poisson process. In a Poisson process, arrivals occur randomly over time, with a constant average arrival rate, denoted by λ (lambda). The time between arrivals follows an exponential distribution. The memoryless property means that the probability of the next arrival does not depend on when the previous arrival occurred.
2. **Service Process (M):** The service times are also memoryless and are typically modeled as an exponential distribution. This is characterized by a constant average service rate, denoted by μ (mu). Like the arrival process, the time until the next service completion is independent of the past.
3. **Number of Service Channels (1):** This indicates that there is only one server in the system. All arriving customers or items must wait in a queue if the server is busy, and they are served one at a time.

The system's stability condition is λ < μ, meaning the arrival rate must be less than the service rate for the queue to not grow indefinitely.

Formulas for performance metrics in a stable M/M/1 queue are derived from λ and μ. For instance, the average number of customers in the system (L) is given by $L = \frac{\lambda}{\mu-\lambda}$, and the average time a customer spends in the system (W) is $W = \frac{1}{\mu-\lambda}$.

This model is widely used in various fields, including telecommunications, retail, and manufacturing, to analyze system performance and make informed decisions about resource allocation and system design.

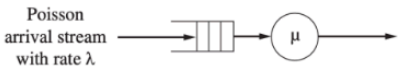
- The birth-death MC can be specialized for

$$\lambda_k = \lambda, \quad k \geq 0; \quad \mu_k = \mu, \quad k \geq 1.$$

$$\pi_k = \left(\frac{\lambda}{\mu}\right)^k \pi_0 = \rho^k \pi_0$$

$$\pi_0 = \frac{1}{\sum_{k\geq 0}\rho^k} = 1 - \rho,$$

$$\pi_k = (1-\rho)\rho^k, \quad k \geq 0.$$

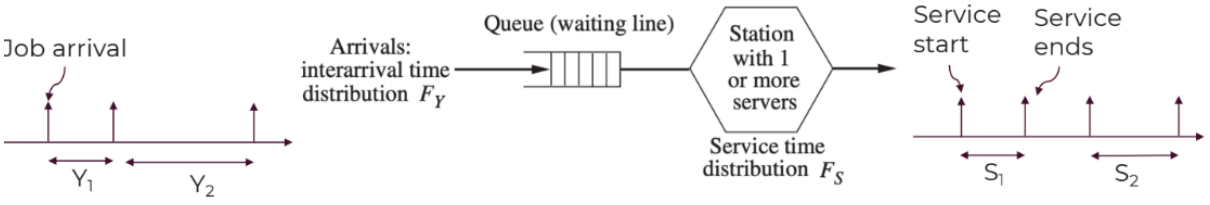


- This is the model of a queueing system where jobs arrives according to a Poisson process and require an exponentially distributed amount of service time (M/M/1)
- The Poisson process is a simple approximation of the aggregate (sum) of independent users that interacts with the service, where a user interacts every T sec (T being memoryless)
- The ratio ρ=λ/μ is called traffic intensity, and it must be <1 for stability

Queueing models: Kendall Notation

- There are others queueing models that are characterized by:

1. interarrival times $Y_1, Y_2, \ldots$, between jobs (independent identically distributed) random variables with distribution F_Y.
2. service times $S_1, S_2, \ldots$, iid random variables having a distribution F_S.
   - The following symbols are used to identify F_Y and F_S:
   - M=Memoryless, aka exponential
   - D=Deterministic
   - E_k=Erlang k-stage
   - H_k=hyper exponential k-stage
   - G=General



3. number of servers, m
4. total capacity of the system, n
5. population size, N
6. scheduling discipline, FIFO, PS, RR,...

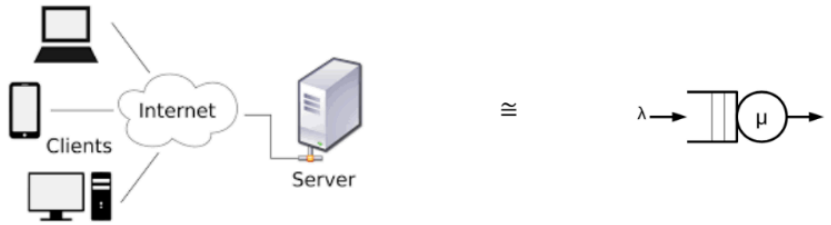
- The Kendall Notation provides a compact description of a queue by means of a 6 fields string **A/B/m/n/N/ω**
- The first three symbols are required, the others by default K=∞N=∞ ω =FIFO

- Useful models for single server
- M/M/1
- M/M/1/n (with capacity n),

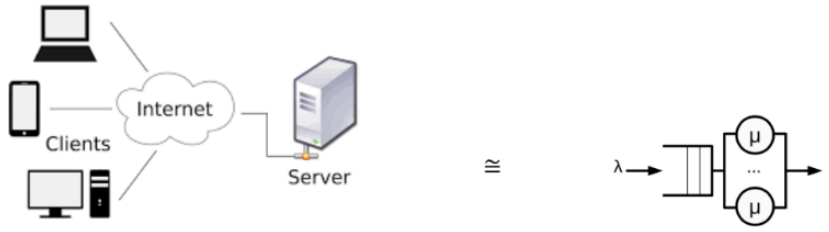
- Models for replicated servers
- M/M/m (with m servers)→ Erlang-C model
- M/M/m/n (with m servers, capacity n>=m)
- M/M/m/m (with m servers, no queuing)→Erlang-B model

- blue = finite system capacity
- red = infinite capacity

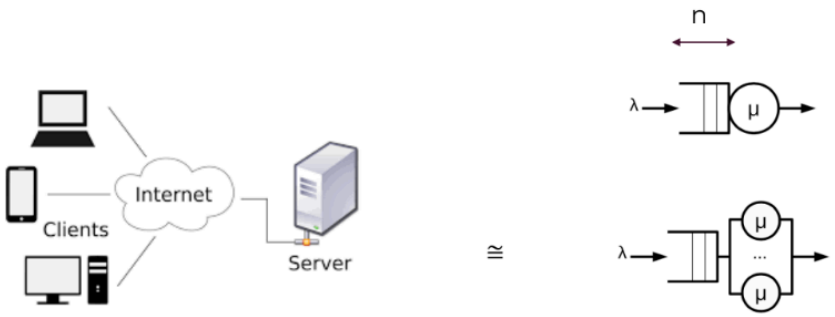
- The simplest performance model for a client/server architecture is the M/M/1 queue
- Despite its simplicity it can provide us useful general insights

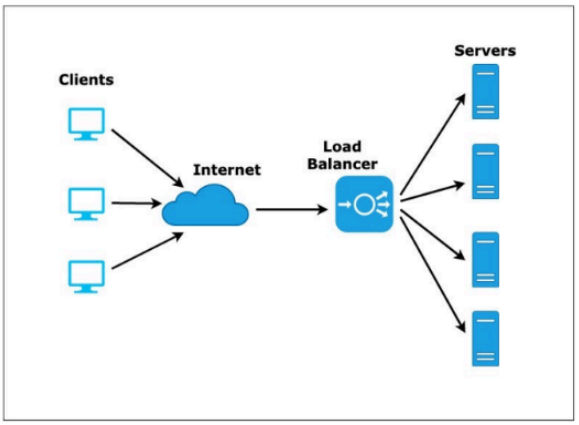


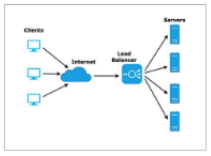
- A variation to the M/M/1 model is to consider M/M/m queue, where n may represent the number of cores of the physical server
- Warning. Name collision server is a word also used in the queue theory

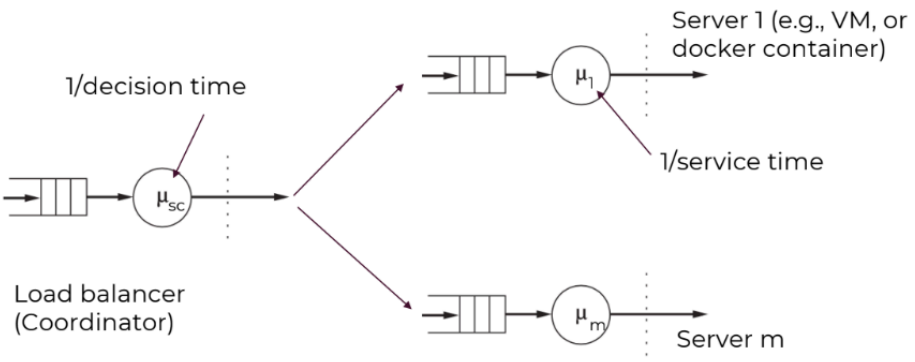


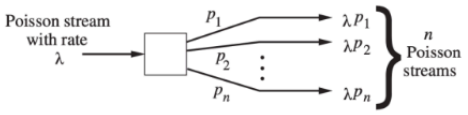
- A variation of the previous models is to consider a finite buffering area
- M/M/1/n or M/M/m/n queues





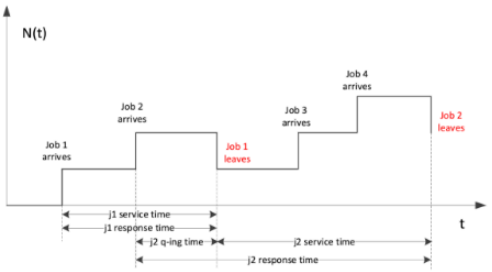


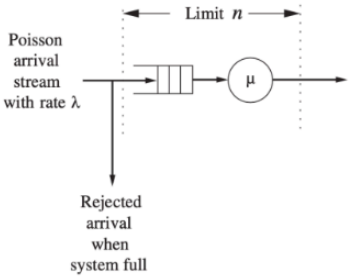




Performance metrics

- Response time
- Loss probability





MM1 Average Response Time

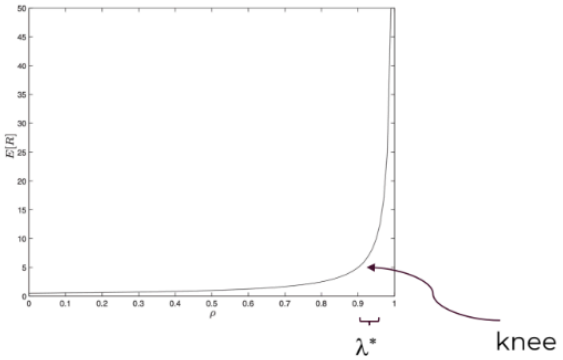
$$E[R] = (E[N]+1)\ 1/\mu = \frac{1/\mu}{1-\rho} = \frac{1}{\mu-\lambda}$$

- Where N is the average number of jobs in the system

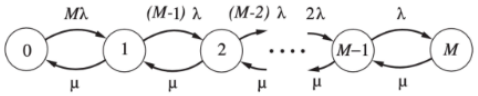
$$E[N] = \sum_{k=0}^{\infty} k\pi_k = \frac{\rho}{1-\rho}$$

- $E[R]$ is the expected response time.
- $E[N]$ is the expected number of customers in the system (both in queue and being served).
- μ is the average service rate (the rate at which the server can serve customers).
- ρ is the utilization of the system, defined as λ/μ, where λ is the average arrival rate of customers.

- There is a 'critical' load after which the response time becomes high...
- For example, assume μ=1
- The response time becomes 10 times the service time when
- 1/1-λ = 10→ λ* = 0.9
- And 20 times the service time when
- 1/1-λ = 20→ λ* = 0.95



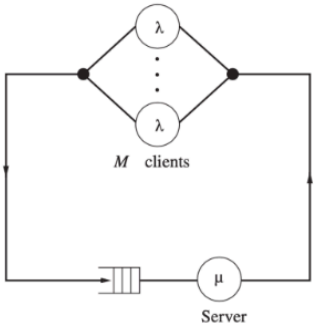
SCALABILITY EXAMPLE 8.13



$$\pi_k = \pi_0 \prod_{i=0}^{k-1} \frac{\lambda(M-i)}{\mu}, \quad 0 \le k \le M,$$

$$\pi_k = \pi_0 \left(\frac{\lambda}{\mu}\right)^k \frac{M!}{(M-k)!} = \pi_0 \rho^k \frac{M!}{(M-k)!}.$$

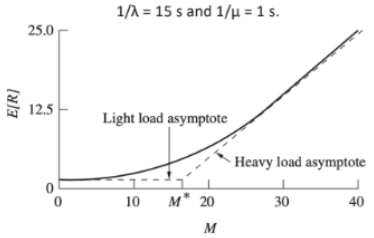
$$\pi_0 = \frac{1}{\sum_{k=0}^{M} \rho^k \frac{M!}{(M-k)!}}.$$

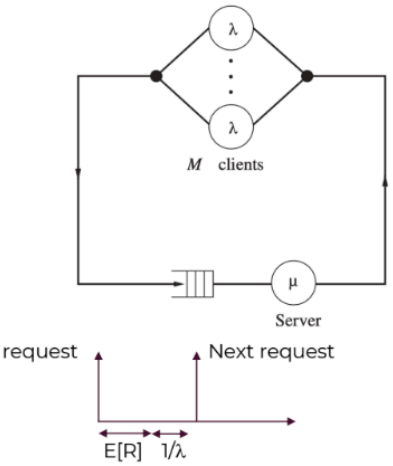


Average exit rate $= \mu(1 - \pi_0)$

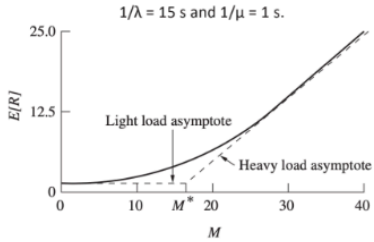
- Average exit rate (completion rate) $= \mu(1 - \pi_0)$
- Average time between two requests of a same client $= E[R] + (1/\lambda)$
- Where E[R]= average response time
- Total request rate from M clients $= M/[E[R] + (1/\lambda)]$

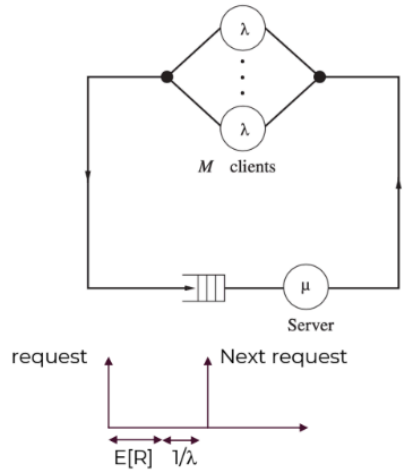
$$E[R] = \frac{M}{\mu(1-\pi_0)} - \frac{1}{\lambda}$$





- The heavy-load asymptote is E[R] = M 1/μ – 1/λ
- The light load asymptote is 1/μ
- They intersect at M*= 1 + λ/μ
- In this example, M*=16

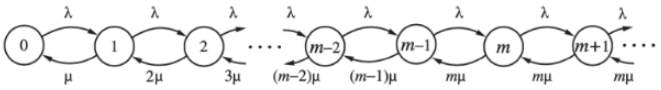


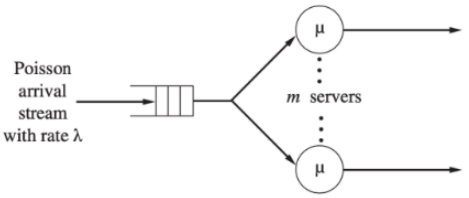


BEYOND AVERAGES

- How these models help answering to the following
- “What is the fraction of clients that are served in at most T ms?”
- “What is the probability that a client is served in more than T ms?”

M/M/m (see 8.2.2 Trivedi's book)





$$\lambda_k = \lambda, \qquad k = 0, 1, 2, \ldots,$$

$$\mu_k = \begin{cases} k\mu, & 0 < k < m, \\ m\mu, & k \geq m. \end{cases}$$

$$\pi_k = \begin{cases} \pi_0 (\frac{\lambda}{\mu})^k \frac{1}{k!}, & k < m, \\ \pi_0 (\frac{\lambda}{\mu})^k \frac{1}{m! m^{k-m}}, & k \geq m. \end{cases}$$

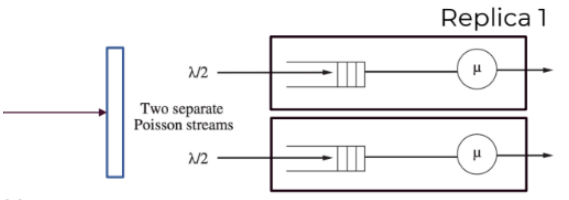
Average Response time: E[R]=E[N]/λ

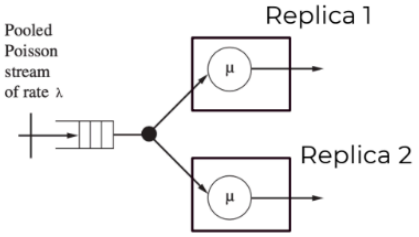
$$E[N] = \sum_{k \geq 0} k\pi_k = m\rho + \rho \frac{(m\rho)^m}{m!} \frac{\pi_0}{(1-\rho)^2}.$$

$$\rho = \lambda/(m\mu)$$

EXAMPLE (FROM EX 8.3 TRIVEDI'S BOOK)

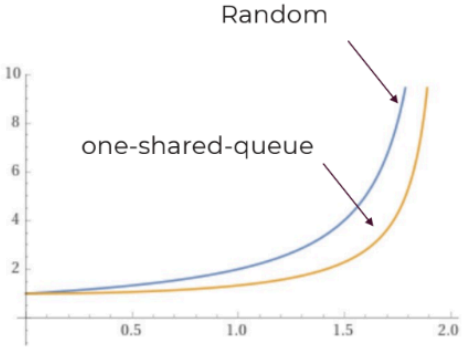
- Let suppose we have to serve a Poisson stream of service requests using a replicated stateless service
- Which design provides the lowest average response time?
- Design 1 (separate): send the requests on their arrivals blindly at random among the two replicas
- Design 2 (common-queue): Enqueue the requests until one of the two servers is idle and then send the request to it





DISCUSSION

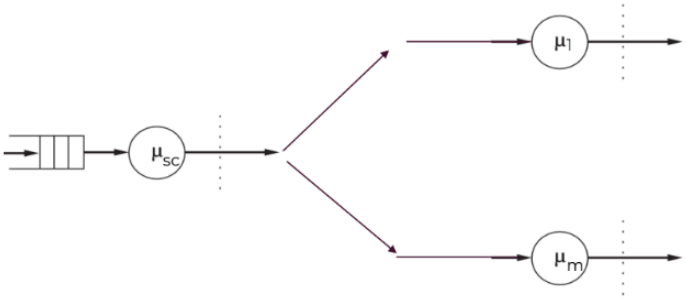
- Separate: $E[R_s] = \dfrac{\frac{1}{\mu}}{1 - \frac{\lambda}{2\mu}} = \dfrac{2}{2\mu - \lambda}.$
- Common: $E[R_c] = \dfrac{4\mu}{4\mu^2 - \lambda^2}.$



μ=1,

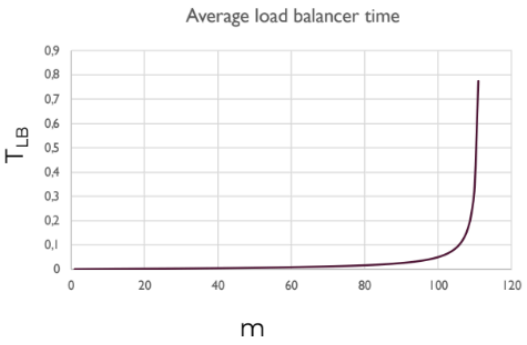
SCALABILITY (SERVER FARM)

- In real life, finding best of m servers is time expensive
- This time T_{sc} is in fact proportional to the number of servers: for example the Load balancer should probe all servers about their state (ping), e.g. $T_{sc}=\alpha m$ (e.g. α = round trip time)
- The Load Balancer time T_{LB} can be estimated using a simple M/M/1 model where $\mu_{sc}=1/(\alpha m)$



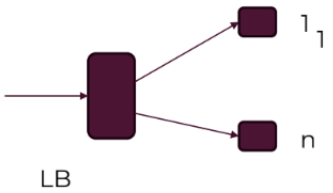
SCALABILITY (SERVER FARM)

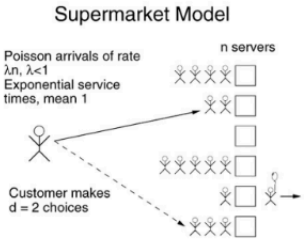
- For example, if $\mu_{sc}=1/0{,}0001$ m and $\lambda=m$ x 0.8 one get the following plot



POWER-OF-2 RANDOM CHOICES

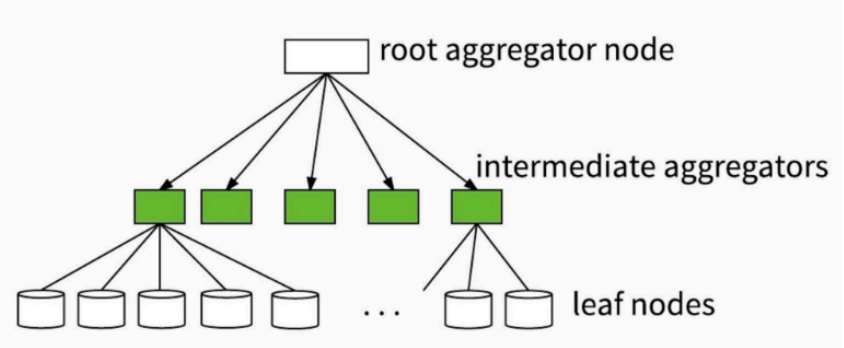
- The Load Balancer (LB) probes just 2 servers at random and selects the best one (generalization to d choices)
- This policy is called Join Shortest Queue among d, JSQ-2, or JSQ-d
- Average response time = $T_{JSQ-d} <= c_d \log T_1$, where T_1, = 1/1-λ (M/M/1 queue) and c is a constant that depends on d, for $\lambda \rightarrow 1^-$ high it is $1/\log_2 d$
- For d=2, high load $T_{JSQ-2} <= \log T_1$
  - Reference paper: *The Power of Two Choices in Randomized Load Balc* Mitzenmacher



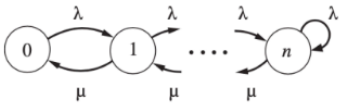


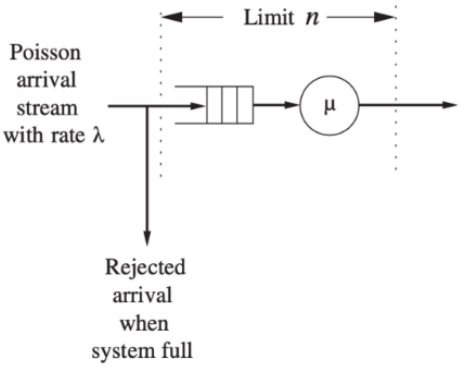
SCALABILITY

- Similar problems arise when one has to query a large number of servers
- There are several solutions to achive scalability.
- For example by using a hierarchiacl design



M/M/1/n





- From the balance equation, $\pi_k=\pi_{k-1}\,\rho$ where $\rho=\lambda/\mu$
- $\pi_k=\pi_0\,\rho^k$
- From $\pi_0+\pi_1+..+\pi_n=1 \rightarrow \pi_0(1+\rho+\rho^2+\rho^n)=1$
- If $\rho<0$, $\pi_0\frac{1+\rho^{n+1}}{1-\rho}=1 \rightarrow \pi_0=\frac{1-\rho}{1+\rho^{n+1}}$
- Reject probability: $\pi_n=\frac{1-\rho}{1-\rho^{n+1}}\rho^n$
- Response time [Little's result]: $E[N]/\lambda(1-\pi_n)$

Little's result is a theorem in the field of queueing theory that relates the average number of customers in a queueing system to the average arrival rate and the average service rate of the system. Little's result states that the average number of customers in a queueing system is equal to the product of the average arrival rate and the average time that a customer spends in the system.

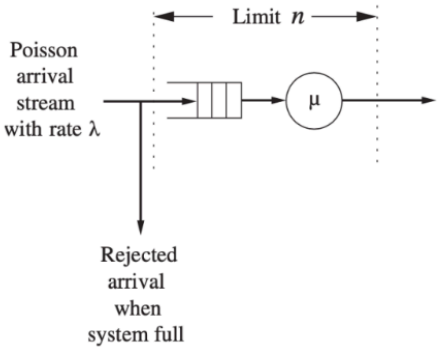
In other words, Little's result can be written as:

$L = \lambda * W$

where L is the average number of customers in the system, λ is the average arrival rate, and W is the average time that a customer spends in the system.

M/M/1/n

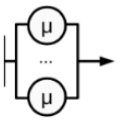
Measure	*Reward rate assignment*	*Expected steady-state reward rate*
Mean number in system	$r_j = j$	$\frac{\rho}{1-\rho} - \frac{n+1}{1-\rho^{n+1}}\rho^{n+1}$
Loss probability	$r_n = 1, r_j = 0 \ \ (j \neq n)$	$\pi_n = \frac{1-\rho}{1-\rho^{n+1}}\rho^n$
Throughput	$r_j = \mu \ \ (j \neq 0), r_0 = 0$ or $r_j = \lambda \ \ (j \neq n), r_n = 0$	$\mu(1-\pi_0) = \lambda(1-\pi_n)$



M/M/n/n

- In this case clients aren't queued and if no servers are available they are not served.
- The probability this event happens, is called the blocking probability and it is given by

$$E_B(m, A) = \frac{\rho^m}{m!}\left(\sum_{i=0}^{m}\frac{\rho^i}{i!}\right)^{-1}$$



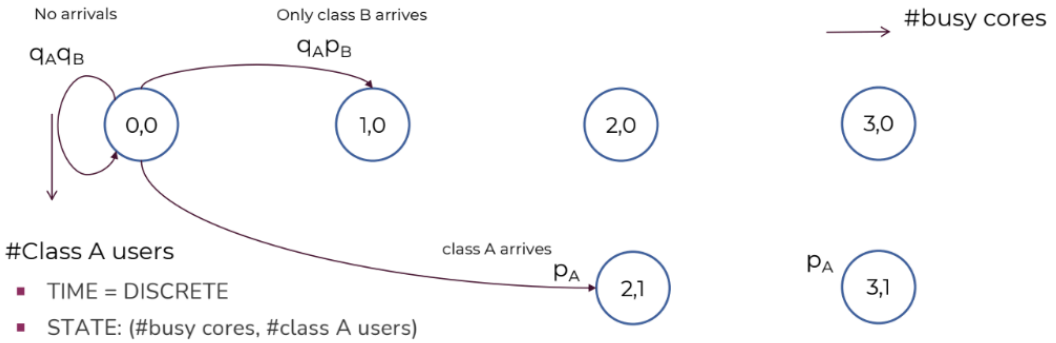
Resource Management using RL

- Two types of users: class A users require 2 cores and class B which require 1 core
- Suppose there are 3 cores in total per physical machine
- The rental cost is proportional to the time of use and the number of cores
- What is the best resource management policy?
- For example:
  - Accept all VM requests
  - Accept only class A requests
  - Accept class B, only if 2 cores remains idle..

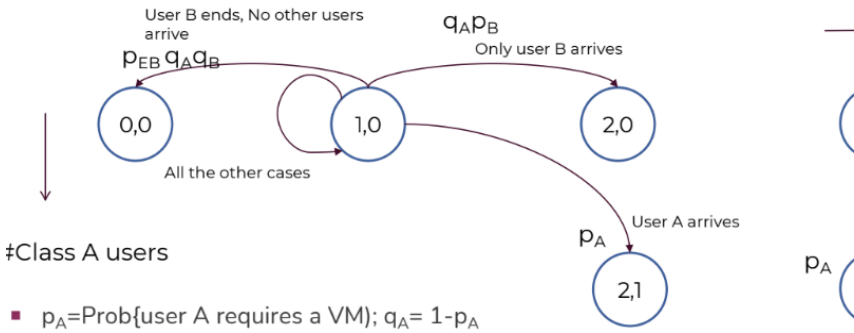
- If we assume Markov property holds, then each policy can be studied by a Markov chain plus rewards
- The Markov chain is specified by a transition probability matrix
- The each policy has its own transition probability matrix

- For example, consider the policy "Accept all VM requests, give priority to class A users in case of contention'

MODELING THE PROBLEM



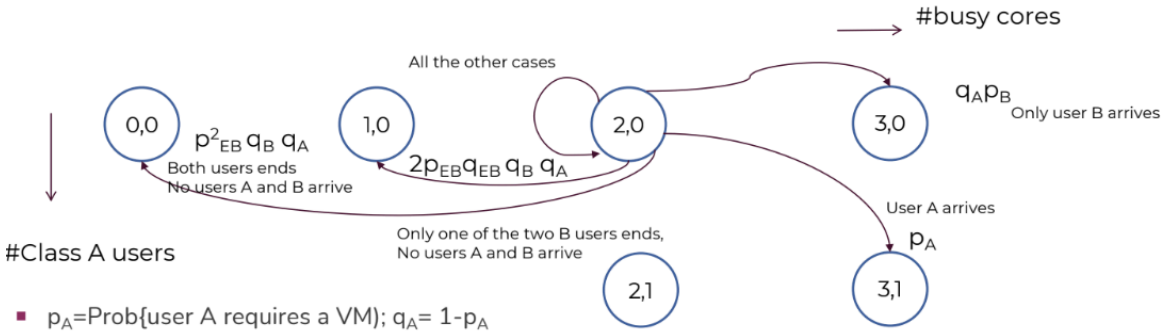
- TIME = DISCRETE
- STATE: (#busy cores, #class A users)
- p_A=Prob{user A requires a VM); q_A= 1-p_A
- p_B=Prob{user B requires a VM), q_B= 1-p_B
- p_{EA}=Prob{user A exits); q_{EA}= 1-p_{EA}
- p_{EB}=Prob{user B exits); q_{EB}= 1-p_{EB}



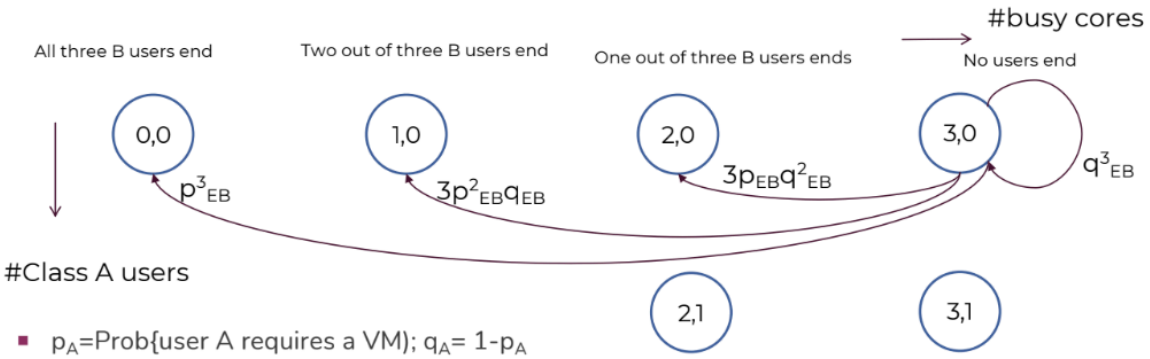
Arrival		Exit	
A	B	A	B
no	yes	*	*
yes	*	*	*
no	no	*	yes

- p_A=Prob{user A requires a VM); q_A= 1-p_A
- p_B=Prob{user B requires a VM), q_B= 1-p_B
- p_{EA}=Prob{user A exits); q_{EA}= 1-p_{EA}
- p_{EB}=Prob{user B exits); q_{EB}= 1-p_{EB}

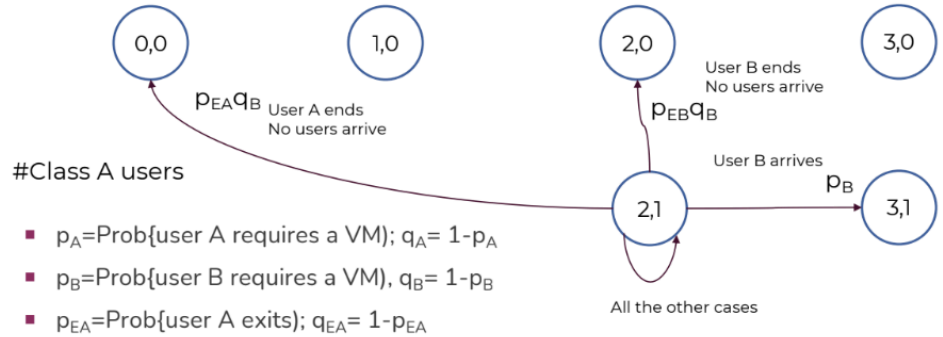
- For simplicity, concurrent events are managed by giving arrivals priority, class A the highest



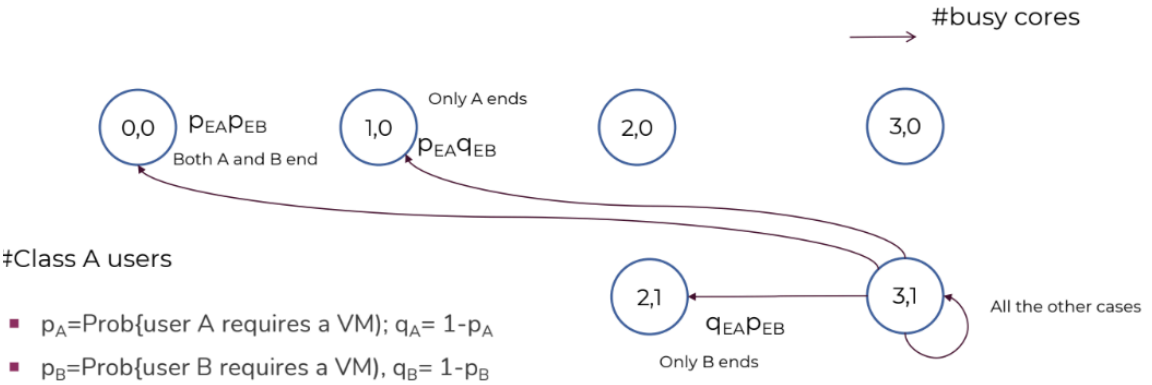
- p_A=Prob{user A requires a VM); q_A= 1-p_A
- p_B=Prob{user B requires a VM), q_B= 1-p_B
- p_{EA}=Prob{user A exits); q_{EA}= 1-p_{EA}
- p_{EB}=Prob{user B exits); q_{EB}= 1-p_{EB}



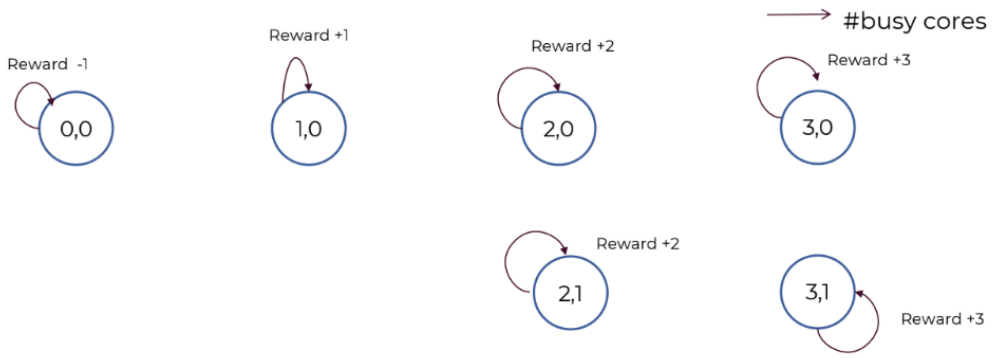
- p_A=Prob{user A requires a VM); q_A= 1-p_A
- p_B=Prob{user B requires a VM), q_B= 1-p_B
- p_{EA}=Prob{user A exits); q_{EA}= 1-p_{EA}
- p_{EB}=Prob{user B exits); q_{EB}= 1-p_{EB}



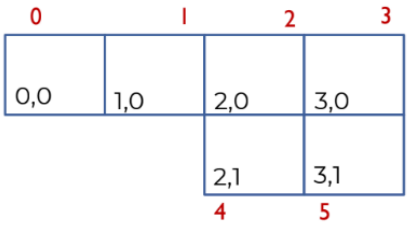
- p_A=Prob{user A requires a VM); q_A= 1-p_A
- p_B=Prob{user B requires a VM), q_B= 1-p_B
- p_{EA}=Prob{user A exits); q_{EA}= 1-p_{EA}
- p_{EB}=Prob{user B exits); q_{EB}= 1-p_{EB}



- p_A=Prob{user A requires a VM); q_A= 1-p_A
- p_B=Prob{user B requires a VM), q_B= 1-p_B
- p_{EA}=Prob{user A exits); q_{EA}= 1-p_{EA}
- p_{EB}=Prob{user B exits); q_{EB}= 1-p_{EB}



- Assign a reward proportional to the number of busy cores, for example just reward=number of busy cores



$$P = \begin{pmatrix} q_A q_B & q_A p_B & 0 & 0 & p_A & 0 \\ p_{EB} q_A q_B & p_{12} & q_A p_B & 0 & p_A & 0 \\ p_{EB}^2 q_B q_A & 2 p_{EB} q_{EB} q_B q_A & p_{22} & q_A p_B & 0 & p_A \\ p_{EB}^3 & 3 p_{EB}^2 q_{EB} & 3 p_{EB} q_B^2 & p_{33} & 0 & 0 \\ q_{EA} q_B & 0 & p_{EB} q_B & 0 & p_{44} & p_B \\ p_{EB} p_{EB} & p_{EA} q_{EB} & 0 & 0 & p_{EB} q_{EA} & p_{55} \end{pmatrix}$$

- To simplify we can label states with integers (the red ones)
- Then we write a reward matrix R

Reward matrix

$$R = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 \end{pmatrix}$$

- How good is this algorithm?
- One approach is to consider the system at its steady-state, i.e. we model a process that runs 'forever'
- Define a total or global reward G as the sum of rewards gained at each step. This sum should be discounted, otherwise it doesn't converge:
- $G=R_1+ \gamma R_2+..+ \gamma^{n-1} R_n+..$
- R_i is the reward due to the i-th transition from the when the system started, for example when 3 cores are used the reward is +3
  - The discount factor $\gamma<1$, in addition to being a "mathematical trick" to guarantee convergence, gives a significant interpretation of the fact that future rewards are "less" important than current ones.
  - For $\gamma=0$, only immediate rewards matter
- The quantity G is indeed a random variable,
- We are interested in its average (which depends on the state from where we start)

- The average global return is the expected return, given that the system started from a state s, namely

$$E[G|S_0=s]$$

- Note: because of the discount factor, even at the the steady state, $E[G|S_0=s]$ depends on s
- The value $E[G|S_0=s]$ then represents the global expected return if we start accumulating the rewards from when the state is s

Markov Reward Process (or model)

MARKOV REWARD PROCESS (OR MODEL)

- A MRP is basically a MC extended with the notion of a reward.
- Formally a MRP is a tuple (S,P,R,γ), where
- S state space
- P Transition probability matrix (irreducible and aperiodic)
- R Reward function, R: SxS → $\mathbb{R}$,
- 0<=γ<1 discount factor

State Space (S): This is a set of all possible states in which the process can be. In the context of an MRP, a state encompasses all the information necessary to predict future states and rewards, assuming the Markov property holds.

Transition Probability Matrix (P): For every pair of states s and s', the matrix contains a probability $P_{ss'}$, which is the probability of transitioning from state s to state s' in one time step. An MRP requires this matrix to be irreducible (every state can be reached from any other state, eventually) and aperiodic (the process does not get trapped in cycles of fixed length).

Reward Function (R): This function $R : S \times S \to \mathbb{R}$ maps each transition from state s to state s' to a real number, which represents the immediate reward received when transitioning between those states.

Discount Factor (γ): The discount factor is a number between 0 and 1 ($0 \leq \gamma \leq 1$) that determines the present value of future rewards. A factor of 0 means the agent is short-sighted and only values immediate rewards, while a factor of 1 means the agent values future rewards as much as immediate rewards.

Utility and the Bellman Equation

The value of being in a particular state is given by the state value function $v(s)$, which is the expected total reward starting from state s, and is called the utility of the state. The expectation is conditioned on the starting state s:

$$v(s) = \mathbb{E}[G|S_0 = s]$$

where G is the total discounted reward from state s.

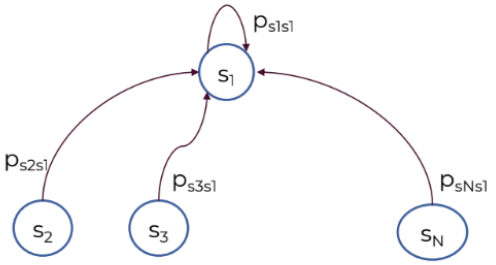
The **Bellman Equation** for MRPs, which arises from the Markov property, is a recursive relationship that helps compute the state value function:

$$v(s) = r(s) + \gamma \sum_{s'} P_{ss'} \cdot v(s')$$

Here, $r(s)$ is the expected immediate reward from state s, defined by:

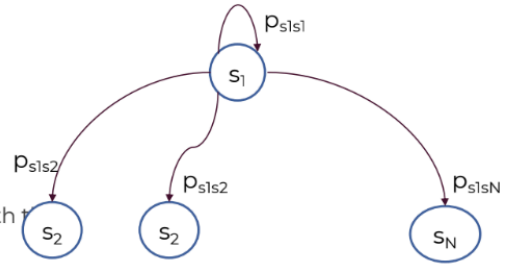
$$r(s) = \sum_{s'} R(s, s') \cdot P_{ss'}$$

- In fact, the Bellman equation connects nearby states that are reachable with some probability
- For example:
- $v(s_1)=r(s_1)+\gamma[p_{s1s1}v(s_1)+p_{s1s2}v(s_2)+p_{s1s3}v(s_3)..\ p_{s1sN}v(s_N)]$
- That is, the value of s_1 is the expected reward plus the values of the reached states (discounted by γ) weighted with the transition probabilities



- A similar relationship between states was used to solve the DTMC using the total probability law (assuming v_{si} is now the steady-state prob of state s_i).
- $v(s_1)=v(s_1)p_{s1s1}+v(s_2)p_{s2s1}+v(s_3)p_{s3s1}+ \dots v(s_N)p_{sNs1}$
- The Bellman equation can be given in matrix form: $\mathbf{v} = \mathbf{r}+\gamma P\mathbf{v}$

$$\begin{bmatrix} v(s_1) \\ \vdots \\ v(s_N) \end{bmatrix} = \begin{bmatrix} r(s_1) \\ \vdots \\ r(s_N) \end{bmatrix} + \gamma \begin{bmatrix} P_{s_1s_1} & \cdots & P_{s_1s_N} \\ \vdots & \ddots & \vdots \\ P_{s_Ns_1} & \cdots & P_{s_Ns_N} \end{bmatrix} \begin{bmatrix} v(s_1) \\ \vdots \\ v(s_N) \end{bmatrix}$$



- $\mathbf{v} = \mathbf{r}+\gamma P\mathbf{v} \rightarrow (I-\gamma P)\mathbf{v}=\mathbf{r} \rightarrow \mathbf{v}=(I-\gamma P)^{-1}\mathbf{r}$
- when γ=1, (I-P) cannot be inverted, (P is stochastic)
  - Recall that In the case of MC, one relationship is replaced with the normalization condition over v
- Similarly to the MC, the value **v** is the fixed point of the iterative equation
- $\mathbf{v}^{k+1} = \mathbf{r}+\gamma P\mathbf{v}^{k}$

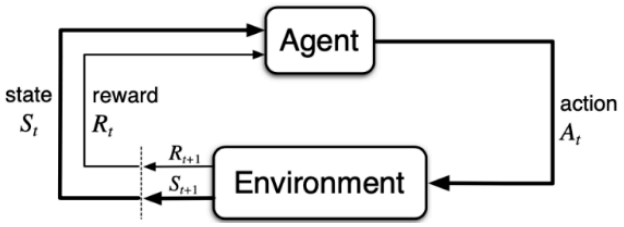
BACK TO THE MOTIVATING EXAMPLE

- What it the best management policy?
- If we know all the transition probabilities then we can solve the MRP and quantify the policy "Accept all VM requests".
- Then, for example, assuming the initial state is given by the steady state probabilities then a single measurement is
- $p_{s1}v(s_1)+p_{s2}v(s_2)+p_{s3}v(s_3)..\ p_{sN}v(s_N)$
- Where p_{si} is the steady state probability of state s_i
- The policy "Accept class B if two cores are idle" has a different transition probability matrix and different value function
- Because the number of policy is finite, one can use brute-force algorithm, i.e. compute all the total returns of the policy and then pick the best one

Markov Decision Process

- What it the best management policy?
- MDP extends MRP by define an abstract general framework that includes:

1. Agent: that decides what to do in each state by making actions
2. Environment: a stochastic Markov model of the all the other things



- A MDP=(S,A,P,R, γ) where
- S state space
- A action space A
- P probability transition function
  - $p(s,s',a)=Pr\{S_{t+1}=s|A_t=a_t,S_t=s'\}$, sometimes denoted as $T(s,s',a)$
- R Reward function, R: SxSxA → $\mathbb{R}$,
- γ discount factor ($0<=\gamma<1$)

1. **State Space (S)**: This is a set of all possible states that the system can be in.
2. **Action Space (A)**: For each state in the state space, there are one or more possible actions that the decision maker can take.
3. **Transition Probability Matrix (P)**: This is a function that determines the probability of moving from one state to another, given a specific action. In an MDP, this function is often denoted as $P(s'|s, a)$, which represents the probability of transitioning to state s' from state s after taking action a.
4. **Reward Function (R)**: This function returns the immediate reward received after transitioning from one state to another, given an action. It is often denoted as $R(s, a)$ or $R(s, a, s')$.
5. **Policy (π)**: A policy is a strategy or rule that the decision maker follows, defined as a function that maps states to actions. Formally, $\pi(a|s)$ is the probability of taking action a in state s under policy π.
6. **Discount Factor (γ)**: Like in an MRP, the discount factor γ determines the present value of future rewards.

Differences Between MDPs and MRPs

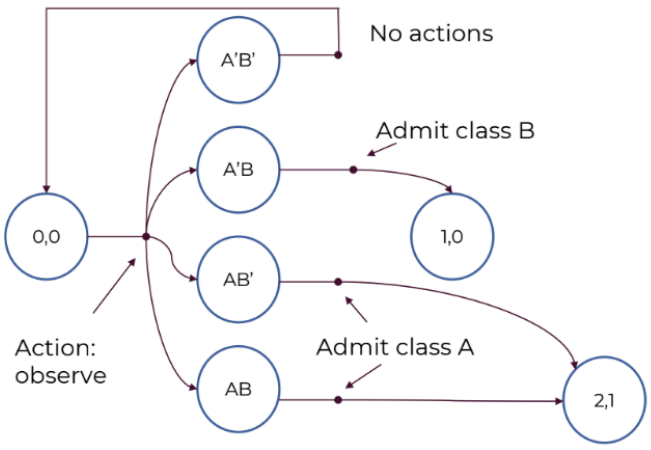
- **Actions**: The most significant difference between an MRP and an MDP is the inclusion of actions in MDPs. While an MRP passively describes the behavior of a system over time, an MDP involves decisions about what action to take at each step.
- **Policies**: Because an MDP includes actions, it also involves policies that specify the best action to take in each state. An MRP has no notion of actions or policies since it is not a decision-making process.
- **Transition Probabilities**: In MRPs, the transition probabilities depend only on the current state. In MDPs, they also depend on the action taken.
- **Reward Function**: In MRPs, the reward function depends on the current state and possibly the next state. In MDPs, it additionally depends on the action taken.

The goal in an MDP is to find an optimal policy π^* that maximizes the expected return (cumulative discounted rewards) from any initial state. This typically involves using algorithms like value iteration or policy iteration to find the value function $V^\pi(s)$ for a given policy π, or directly finding the optimal value function $V^*(s)$ and the optimal policy π^*.

The Agent-Environment interface

- The environment is a memoryless entity, i.e. the change depends only its current state and not on the history (how it reached that state)
  - Comment: we can add info to the state like, the state is the last two arrivals ☺
- The reward R_{t+1} is what the agent gets when going from s to s' due to action A_t
- We assume that the MC behind this model is stationary

BACK TO THE EXAMPLE



- to use MDP we can for example assume that our agent wakes up periodically to observe the incoming requests, add 4 states that represent

1. Neither class A nor class B arrivals (A'B')
2. Only class B (A'B)
3. Only class A (AB')
4. Both class A and B (AB)

And then apply the policy

Usually actions are represented as smaller filled circles

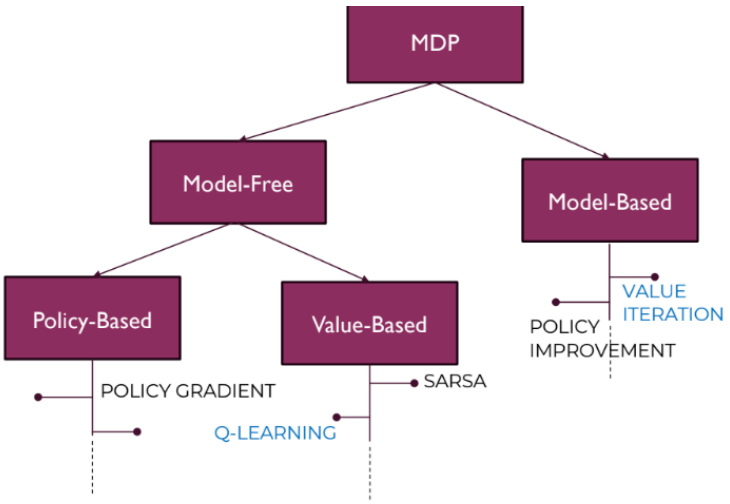
As the reaction to an action is deterministic, these states can be 'implicit'

Policy

- A policy is a rule π to determine which action a to make in a given state s, it can be stochastic or deterministic
- The key advantage of the MDP is that it allows to formalize the notion of optimal policy, which is the policy that maximizes the expected reward, G
- In addition it is the theorical framework for Reinforcement Learning (RL) algorithms that 'discover' the optimal policy autonomously and without a defined probability transition model

Solution of an MDP

- Determining the best policy is said to solve a MDP
- Because MDP formulation is the foundation of RL, to determine the optimal solution is also known as solving a RL task
- There are several way to find the best policy
- A coarse difference is if the model is given (Model-based) or not (Model-Free) and the algorithm adopted for learning



Model-Based Reinforcement Learning:

Model-based RL methods involve an agent that builds a model of the environment's dynamics. This model includes the transition probabilities (how likely it is to end up in a new state given the current state and an action) and the reward function (the immediate reward received after taking an action in a state). The agent uses this model to plan and decide on the best actions by simulating future states and rewards.

An example of model-based RL is using a decision tree, dynamic programming, or planning algorithms like Monte Carlo Tree Search (MCTS).

Model-Free Reinforcement Learning:

Model-free RL methods do not build a model of the environment. Instead, they learn to make decisions based solely on the observed rewards and states from interaction with the environment. These methods directly learn a policy or a value function without needing to know the transition probabilities or reward function.

Examples of model-free RL include Q-learning, SARSA (State-Action-Reward-State-Action), and policy gradient methods like REINFORCE.

State value function

In the context of a Markov Decision Process (MDP), the state value function, often denoted as $V(s)$, represents the expected return (total accumulated rewards) when starting from state s, and thereafter following a particular policy π. The return is usually calculated as the sum of rewards the agent expects to receive in the future, which are discounted by a factor of γ at each time step to account for the uncertainty of future rewards.

Formally, the state value function for a policy π is defined as:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, \pi\right]$$

where:

- $V^{\pi}(s)$ is the value of state s under policy π.
- $\mathbb{E}$ denotes the expected value, indicating that $V^{\pi}(s)$ is an expectation based on the randomness in the policy and the stochastic nature of the environment.
- R_{t+k+1} is the reward received after $k+1$ time steps.
- $S_t = s$ means the agent is in state s at time t.
- γ is the discount factor, $0 \leq \gamma \leq 1$, which determines the present value of future rewards—a smaller value of γ will place more emphasis on immediate rewards.
- π is the policy being followed, a mapping from states to probabilities of selecting each possible action.

The value function thus captures the "goodness" of states, given the policy π. If an agent follows policy π, $V^{\pi}(s)$ provides the expected amount of reward the agent can expect to accumulate over the future, starting from state s.

- The state-value function obeys to the following equation (Bellman) [for the sake of simplicity the policy here is deterministic]

$$v^{\pi}(s) = \sum_{s'} p(s, s', \pi(s))[r(s, s', \pi(s)) + \gamma v^{\pi}(s')]$$

Let's decipher this equation:

- $v^{\pi}(s)$: The value of being in state s under a policy π.
- $\sum_{s'}$: A sum over all possible next states s'.
- $p(s, s', \pi(s))$: The probability of transitioning from state s to state s' when action $\pi(s)$ is taken. This is part of the model of the environment.
- $r(s, s', \pi(s))$: The immediate reward received after transitioning from state s to state s' due to action $\pi(s)$.
- γ: The discount factor, which determines the present value of future rewards. It makes sure that rewards received sooner are worth more than rewards received later.
- $v^{\pi}(s')$: The value of being in the next state s', again under policy π.

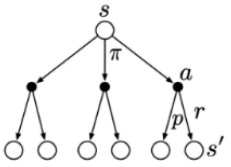
The equation states that the value of a state under a deterministic policy is the sum of the expected rewards for each possible next state, plus the discounted value of being in that next state, weighted by the probability of reaching that next state through the policy's chosen action.

The deterministic policy $\pi(s)$ maps states to actions without any randomness; that is, given a state s, the policy π will always choose the same action.

This Bellman equation is fundamental in dynamic programming approaches to solving MDPs, like value iteration and policy iteration, where it is used iteratively to compute the value of each state until the values converge to a stable solution. The converged values can then be used to derive the optimal policy.

- The Bellman equation is illustrated by a graph where each open circle represents a state and each solid circle represents a state–action pair. Starting from state s, the root node at the top, the agent could take any of some set of actions— three are shown in the diagram— based on its policy π.
- From each of these, the environment could respond with one of several next states, s' (two are shown in the figure), along with a reward, r, depending on its dynamics given by the function p.

$$v^{\pi}(s) = \sum_{s'} p(s, s', \pi(s))[r(s, s', \pi(s)) + \gamma v^{\pi}(s')]$$

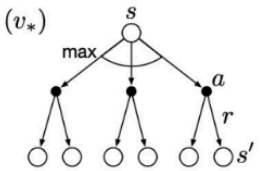


Optimal state value function

- A policy π is defined better than or equal to a policy π' if $v_{\pi}(s) >= v_{\pi'}(s)$ for all s
- The optimal state-value function is $v^*(s) = \max_{\pi} \{v_{\pi}(s)\}$
- The optimal state-value obeys to the Bellman optimality equation

$$v^*(s) = max_a \sum_{s'} p(s, s', a)[r(s, s', a) + \gamma v^*(s')]$$

- If one knows $v^*(s)$ then the best policy is just the action that maximizes the future expected global return



VALUE ITERATION

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

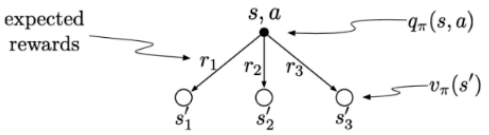
Quality function

- A closely related function is the quality or state-action function, or Q function, defined as

$$Q^\pi(s, a) \doteq E_\pi[G|S = s, A = a]$$

- This function is the expected return starting from s, taking the action a, and thereafter following policy π

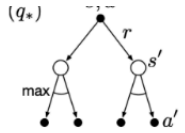
$$Q^\pi(s, a) = \sum_{s'} p(s, s', a)[r(s, s', a) + \gamma Q^\pi(s', \pi(a)]$$



- $\sum_{s'}$ is the sum over all possible next states s'.
- $p(s', s, a)$ is the probability of transitioning from state s to state s' after taking action a.
- $r(s, s', a)$ is the immediate reward received after taking action a in state s and transitioning to state s'.
- γ is the discount factor, which represents the difference in value between immediate and future rewards.
- $Q^\pi(s', \pi(s'))$ is the expected return from the next state s' following the current policy π.

- The optimal state-action function is given by

$$Q^{\pi^*}(s,a) = r(s,s',a) + \sum_{s'} p(s,s',a)V^{\pi^*}(s)$$



- If the agent is in state s, then the best action to do is

$$a^* = argmax_a Q^{\pi^*}(s,a)$$

- In fact in this way, the agent gets the maximum value associated to the state s
- In other words, given Q* it is straightforward to derive on optimal policy:

$$\pi^*(s) = argmax_a Q^*(s,a)$$

- In case of ties, the actions are equivalent (just pick one action at random)
- One way to compute the optimal Q function is (again) through iterations (note there is no need to explicitly reference a policy, this is the Q* of the MDP)

$$Q^*(s,a) = \sum_{s'} p(s,s',a)[r(s,s',a) + \gamma max_{a'} Q^*(s',a')]$$

Temporal difference and Q-Learning

- Q-Learning is an algorithm that allows to compute Q* without to know the model of the environment
- If the state space is small, then the function Q* is just a table
- When the state space grows, the function Q* is approximated via neural networks and the algorithm is called Deep Q-learning
  - This also allows to 'generalize' the behaviour of the agent to include states never visited.
- Q-learning is based of temporal difference, which is a key breakthrough in RL
- "If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning." (Sutton-Barto)

- The key ideas of Q-learning are
- Learn from data: adapt the policy using the reward as feedback
- Bootstrap: adapt an estimation, using estimations of other correlated quantities, that of course are not precise, or it builds estimations based on other estimations
- Off-policy: Learn the best policy directly. For this purpose, it uses a mix of exploitation – i.e. what was learned until now - and exploration of new actions (in fact off-policy actions)

- The goal is maximizing the reward assuming that there is a finite number of steps (episodic tasks)
- In episodic task γ can be 1

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state,\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Take a random action with probability ε

Similar to the rule $\mu \leftarrow \mu + \alpha(x-\mu)$ ..but x is drawn from an estimation itself

Figure 6.12: Q-learning: An off-policy TD control algorithm.

This is like a sample of the random variable $X=r(s,s',a)+\gamma \max_{a'} Q^*(s',a')$

$$Q^*(s,a) = \sum_{s'} p(s,s',a)[r(s,s',a) + \gamma max_{a'} Q^*(s',a')]$$