GFS

- **Overview**: Large Files(100MB+); optimized reads+appends;high and consistent bandwidth, commodity machines, sequential reads.
- **Master**: Stores metadata in-memory, including file and chunk namespaces (persistently), chunk handles for each file, location of each chunk's replicas (in-memory); Queries all chunk servers to see what chunks they have at startup; **Master Failing?** Shadow masters take over serving requests when the master dies; Master stores an operation log to recover state on failure.
- **Chunk servers** store parts of files - chunks in 64MB blocks, which are identified by a 64-bit chunk handle.
- **Consistent Data:** all clients see the same data, regardless of which replicas they read; **Defined Data:** data is consistent and clients see mutations in their entirety; GFS guarantees that after a number of mutations (writes or appends), the modified file will be defined
- **Deletion**: no immediate reclamation/deletion; rename files to hidden names; during chunk scans by master, hidden files get removed and space reclaimed
- **Copy-on-Write:** Snapshots are made lazily, copying data only on modification/writes. Minimizes data duplication and reduces the overhead associated with creating snapshots.
- **Atomic Append:** Appends the data atomically as a continuous sequence of bytes. Useful for concurrent writes by multiple clients to the same file. Replicas may have slightly different data, including potential duplicates, but guarantees at-least-once appends.
- **Fault Tolerance**: Master is replicated with shadows that allow for read-only if master fails with possibly slightly old metadata; Master gets monitored externally and restarted on failures; chunk servers come and go; Master is responsible for cloning replicas as needed to maintain high availability; Checksums every 64KB to detect integrity issues. No need to check across replicas.

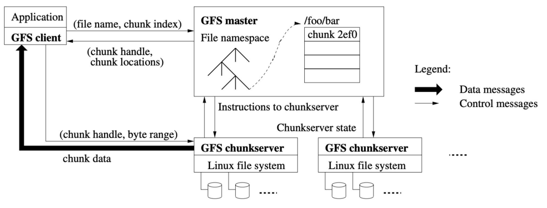


Figure 1: GFS Architecture

Misc.

IaaS - renting hardware on demand; responsible for everything Ex) AWS EC2, GCP, GCE, Azure, Virtual Machines
PaaS - environment; responsible for apps and data Ex) GCP, AppEngine, Kubernetes, Docker
SaaS - everything provided, bring the data Ex) Salesforce, Gsuite, Office365, Vision, NLP, Analytics and other services
REST Pros: **Scalability**: handles multiple requests. **Cacheability**; **Simplicity and Flexibility**: HTTP methods
REST Cons: **Stateless Constraints**: extra work for states. **Performance Overhead**: methods introduce network overhead. **Limited Methods**: Standard methods limiting for complex operations.
RPC- protocol to request a service from a program located elsewhere in a network.
Pros: Simplicity; **Efficiency**: More efficient than REST for complex ops. **Language Independence**
Cons: **Tighter Coupling**: Requires knowledge of the procedures available. **Complexity in Debugging and Testing**: harder to trace and debug. **Firewall and Security**: extra work for firewalled configs.
Pub/sub Pros: Decoupling: Pub/Sub are loosely coupled for flexibility and scalability. **Asynchronous Communication**: Allows high-volume, high-throughput message streams. **Flexibility in Processing**: Messages processed as needed.
Pub/sub Cons: Complexity in Managing Subscriptions; Message Delivery Guarantees; **Dependency on Broker**: Relies on a central broker, which can be a single point of failure.
Comparative Analysis: **Decoupling:** Pub/Sub most decoupled, RPC tighter. **Communication Pattern**: REST/RPC request-response based, Pub/Sub publish-subscribe. **Scalability**: REST & Pub/Sub more scalable due to stateless **Suitability**: REST: APIs, Pub/Sub: event-driven arch, and RPC: service-to-service.
IP- **Role**: addressing **Dependency**: **On TCP/BGP**: TCP for delivery and BGP for routing **On Load Balancers**: used for directing traffic.
TCP - **Role**: delivery. **Dependency**: **On IP**: needs to know dest **On Load Balancers**: manages and optimises.
BGP - **Role**: routing **Dependency**: **On IP**: dest and routing choices. **On TCP**: establish connection.
TCP vs UDP - TCP reliable/state; UDP connectionless/stateless
Routing and Communication Flow: **IP** assigns addr to devices **TCP** ensures data is delivered reliably and in order, builds on IP. **BGP** for routing using the IP protocol.
IP Addressing & Subnetting: Fundamentals for identifying devices and segmenting networks logically.
VPN: Creates secure connections over public networks.
(Layer 1) Physical layer - fiber, wireless, hub, **(Layer 2)** Data link - frame like the ethernet, switch, **(Layer 3)** Network - packets, IP, ICMP, **(Layer 4)** Transport - end to end connection via UDP, TCP, **(Layer 5)** Session - synch and send to port, APIs websockets, **(Layer 6)** Presentation - syntax layer, SSL, SSH, **(Layer 7)** Application - end user layer, HTTP, FTP

Kafka

- **Overview**: Pub/Sub system with **topics**, **producers**, and **consumers**. Published messages are stored at **brokers**, consumers subscribe to and pull messages from brokers.
- Consumers can have **one or more streams** for a topic. Messages will be **evenly distributed** in these streams. Topics are divided into multiple **partitions** across multiple brokers. Brokers can have **more than one partition** for each topic. Messages might be **consumed by multiple** subscribers.
- Brokers are stateless and have a **7 day** retention time on messages. Consumers can **rewind** to older times. **No master** node - consumers figure it out in a decentralized fashion
- **Zookeeper:** Detects the addition and removal of brokers and consumers, triggers a **rebalance** process in each consumer. Keeps track of the consumed offset of each partition. If a broker fails, all partitions on it are automatically removed from the broker registry. **All unconsumed data is lost.**

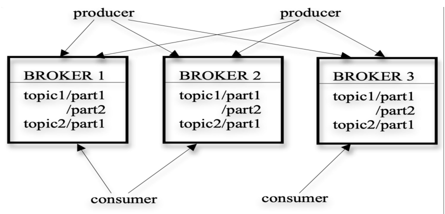


Figure 1. Kafka Architecture

```
Algorithm 1: rebalance process for consumer $C_i$ in group G
For each topic T that $C_i$ subscribes to {
  remove partitions owned by $C_i$ from the ownership registry
  read the broker and the consumer registries from Zookeeper
  compute $P_T$ = partitions available in all brokers under topic T
  compute $C_T$ =  all consumers in G that subscribe to topic T
  sort $P_T$ and $C_T$
  let j be the index position of $C_i$ in $C_T$ and let $N = |P_T|/|C_T|$
  assign partitions from $j*N$ to $(j+1)*N - 1$ in $P_T$ to consumer $C_i$
  for each assigned partition p {
    set the owner of p to $C_i$ in the ownership registry
    let $O_p$ = the offset of partition p stored in the offset registry
    invoke a thread to pull data in partition p from offset $O_p$
  }
}
```

- **Logging**: Each partition is a **logical log,** a sequence of segment files. Messages are addressed by their offset in the log. Each broker keeps a **sorted list of offsets**, including the offset of the first message in every segment file.
- By keeping data in logs, partitioning logs across brokers enables horizontal scaling-increase capacity by adding more brokers to the cluster, allowing handling more producers and consumers and larger data volumes. Consumers can track the offset to maintain state of what messages have been consumed.

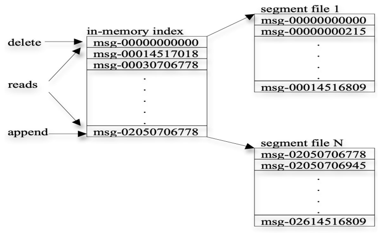
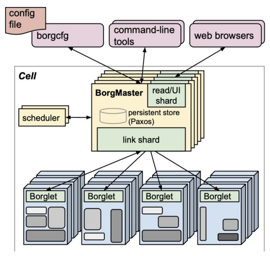


Figure 2. Kafka log

HTTP Methods: GET HEAD: Header info only, POST: Send data, PUT: Replace resource, DELETE: Remove resource, CONNECT, OPTIONS, TRACE: Specialized methods
Status codes:1xx: Informational, 2xx: Success, 3xx: Redirect, 4xx: Client Error, 5xx: Server Error
HTTP: Basic web protocol without encryption, **TLS**: Provides security for HTTP, making it HTTPS, **HTTPS**: HTTP over TLS/SSL, used for secure communication, **SSL**: Outdated version of TLS (not recommended)
Routing table for a network: includes the network itself and immediate neighbors
IPv4: 32-bit length, Structure: Four 8-bit octets
IPv6: 128-bit length, Structure: Eight 16-bit blocks
Network Load Balancers - On TCP/UDP level, making decisions from IP and port. Depends on IP and TCP **Traffic Type**: Any IP traffic, **Complexity**: Low, **Deployment**: Regional, **Balancing Ability**: Coarse
HTTP Load Balancers - On application layer, distributes based on content type, headers, etc. Relies on TCP and IP. **Traffic Type**: HTTP only, **Complexity**: High, **Deployment**: Global, **Balancing Ability**: Fine

Borg

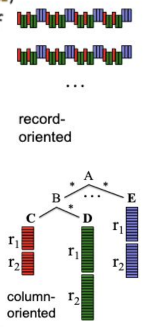
- 3 main components - **borgmaster**, **borglets**, **borg config**
- **Borgmaster:** Collection of 5 machines, handles scheduling of jobs to machines.
- Tasks are divided as prod and non-prod;4 levels - monitoring, prod, test and batch. Cannot preempt prod in favor of prod job;
- **Borglet**: Lives on each worker, interacts with the kernel for scheduling decisions, communicates with borgmaster. Size of a cluster depends on power; Heterogeneous machines
- Security: chroot jails and VMs to prevent tasks from interfering with one another; **performance:** isolations via separate cpus, and linux cgroups.
- **Feasibility checking**: check if the machine can meet task constraints and have enough available resources. **Scoring**: determine the goodness of job-evenly utilize all resources in a machine, check if machines already have necessary packages.
- CPU: ~70% allocation, ~60% usage
- Memory: ~55% allocation, ~85% usage.
- Job has priorities and quotas. Lower priority jobs have infinite quotas for the sake of preventing oversubscription of higher priorities.
Optimizations:
Score caching - keep track of scores in a cache
Equivalence class - similar problems have the same score
Choose a subset of machines instead of all of them



Dremel

Row Storage: Stores records side-by-side, with all columns for a single record grouped together; Efficient for fetching entire records and frequent individual row updates; Can be less efficient for queries involving columns across many rows.
Column Storage: Stores each column's data continuously; Efficient for queries involving specific columns across many rows; Better compression due to data type homogeneity with columns; Less efficient for fetching entire records and frequent individual row updates. Similar data types within columns compress better than diverse data types within rows.
Dissecting Record -> Columns **Identify the record format:** Understand the delimiter (comma, tab, etc.) or fixed-width structure. **Split the record based on the format. Assign each field to its corresponding column:** Store each extracted value in its appropriate column variable or data structure.
Repetition level tells which repeated field this value belongs:
0 -> New record
1-> 1st repeated field
2 -> 2nd repeated field
Null values are inserted when a repeated field appears 0 times.
Definition level tells us which fields at that nesting depth that are repeated or optional are actually present:
0 -> Required top level field
1 -> at first nesting level of optional/repeated
2 -> at second nesting level of optional/repeated
3 -> at third nesting level of optional/repeated

```
procedure DissectRecord(RecordDecoder decoder,
            FieldWriter writer, int repetitionLevel)
  Add current repetitionLevel and definition level to writer
  seenFields = {} // empty set of integers
  while decoder has more field values
    FieldWriter chWriter =
      child of writer for field read by decoder
    int chRepetitionLevel = repetitionLevel
    if set seenFields contains field ID of chWriter
      chRepetitionLevel = tree depth of chWriter
    else
      Add field ID of chWriter to seenFields
    end if
    if chWriter corresponds to an atomic field
      Write value of current field read by decoder
      using chWriter at chRepetitionLevel
    else
      DissectRecord(new RecordDecoder for nested recor
        read by decoder, chWriter, chRepetitionLevel)
    end if
  end while
end procedure
```



MapReduce:

Fault Tolerance MapReduce achieves fault tolerance by periodically creating checkpoints and by re-executing tasks in case of worker failures; **Master Failure:** A secondary master takes over if the primary master fails; **Worker Failure:** Incomplete tasks are re-executed on other available workers; Any complete or in progress map or in progress reduce tasks are reset to idle and get rescheduled; Completed reduce tasks write their output to the persistent storage. **Reason:** This redundancy ensures that the system can recover from both hard failures (e.g. machine crashes) and soft failures (e.g., slow performance due to resource contention)

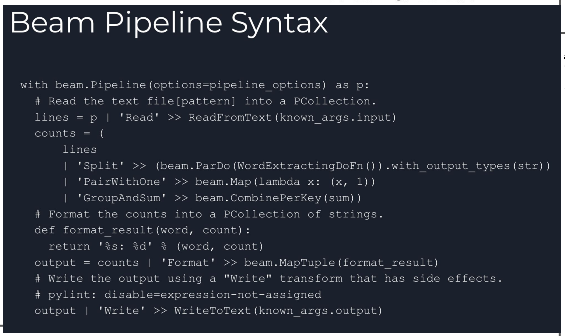
Time Consumption **Hard Failures:** Recovery from machine crashes or other critical failures takes time. **Soft Failures:** Redundant execution of tasks is essential for fault tolerance but may lead to longer overall execution times, especially in the presence of slow-performing machines.

Operations **Map:** Performs independent operations on individual key-value pairs. **Reduce:** Groups intermediate key-value pairs and applies an aggregate function to each group.

Parallelism **Map:** High, due to independent tasks. Multiple workers can process input data simultaneously. **Reduce:** Lower, limited by number of unique keys. Multiple workers can handle different key groups, but not the same key simultaneously.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```



Beam Pipeline Syntax

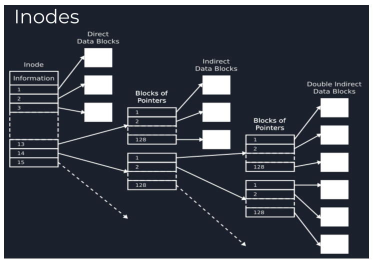
```
with beam.Pipeline(options=pipeline_options) as p:
  # Read the text file[pattern] into a PCollection.
  lines = p | 'Read' >> ReadFromText(known_args.input)
  counts = (
      lines
      | 'Split' >> (beam.ParDo(WordExtractingDoFn()).with_output_types(str))
      | 'PairWithOne' >> beam.Map(lambda x: (x, 1))
      | 'GroupAndSum' >> beam.CombinePerKey(sum))
  # Format the counts into a PCollection of strings.
  def format_result(word, count):
    return '%s: %d' % (word, count)
  output = counts | 'Format' >> beam.MapTuple(format_result)
  # Write the output using a "Write" transform that has side effects.
  # pylint: disable=expression-not-assigned
  output | 'Write' >> WriteToText(known_args.output)
```

Normal Forms:

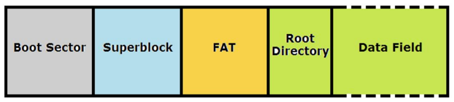
1NF - each attribute has **only one value**; all attribute values can't be divided into something smaller
2NF - satisfies 1NF; **no non-key attribute depends on subset of key**, must depend on entirety of key
3NF - satisfies 2NF; **all non-key attributes depend on key and nothing else**
Indexes: way to efficiently find rows that have attributes meeting constraints; primary keys are always index; Indices on other attributes or attribute combinations can be added;
Clustered index - defines order in which data is PHYSICALLY stored in table; fast; **unclustered index** - like a table of contents; index is stored in one place and data is stored in another; slow;
Hash indices - good for finding unique values; bad for collection of values or ranges; **B tree indexes** - good for doing a range query; **R tree indices** - good for spatial query (2-D)

Containers vs VMs
- VMs virtualize the hardware and allow you to run many different OSes on top of the Hypervisor;Containers virtualize the OS and allow you to run many different libraries and middleware on top of a single Host OS;
Container Pros: Iteration speed, good support;
Cons: Security, Multi-OS apps; eg - Docker, Podman, RKT;
VM Pros: Full isolation/security, iterative development;
Cons-Expensive to update; eg-VirtualBox, VMWare, KVM, Xen;

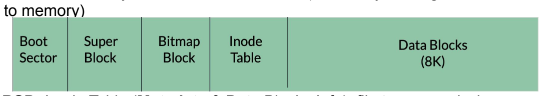
Namespaces + CGroups = Containers.
CGroups: **allocate, prioritize, deny, manage, and monitor system resources**. Used to ensure that a single process or a group of processes doesn't consume excessive resources.
Ex) **PID,** cpuset, cpu, cpuacct, io, memory, devices, freezer, net_cls, net_prio, perf_event, hugetlb -
Namespaces: a mechanism that allows programs that live inside it to use **names that are unique to the underlying OS** (even if overlapping with names from other namespaces).
Ex) **PID**, Network, Mount, User, IPC, UTS



File Systems:



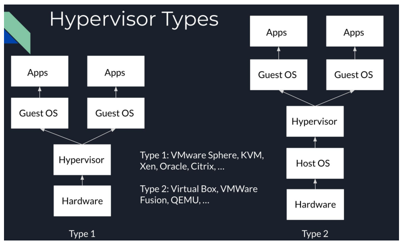
FAT: Superblock (**Metadata**): root directory, size of system, first block
Pros: Arbitrary file sizes (chain of pointers can be arbitrarily long); Limited fragmentation (last block of a file may not be full); **Cons:** Easy to become inconsistent; Very slow for random access (flexible by moving the FAT table to memory)



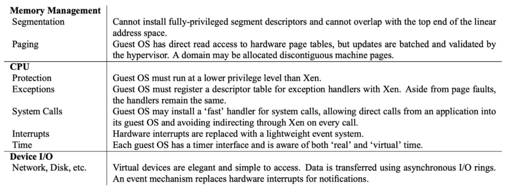
BSD: Inode Table (**Metadata** & Data Blocks Info): file type, permissions, dates/timestamps, links, **direct/indirect/double indirect blocks**
Pros: Random access is easy (all info in the inode, or in easily discoverable pointer blocks); Grow files quickly/efficiently; Easy renaming/copying/linking;
Cons: Some fragmentation is still possible; Number of inodes fixed at file system creation
Journaling File System **Possible corruption during power or other hardware failure while doing file operations (e.g. delete file):** Remove directory entry; Release inode to pool of free nodes; Return disk blocks to pool of free blocks;**Avoid corruption by:** Log all future operations to a journal; When operations are done mark the journal as completed; On a crash incomplete journal operations are replayed to make file system consistent **ext4** **Key improvements:** 48 bit block numbers (1Ebyte file system size); Extents to represent multiple contiguous data blocks (for large files). Up to 2^15 blocks in a single extent; 4K blocks instead of >8K data blocks for less fragmentation; Dynamic instead of static inode allocation; Preallocation of blocks for expected large files; Defragmentation of large files; Reliability improvements

Xen

Type 1 Hypervisor: Runs directly on hardware to host OSes.
Type 2 Hypervisor: Runs on top of a host OS.



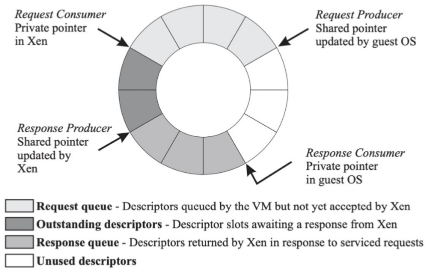
Overview: Requires modified OSes to work on top of Xen (ex. Ring permissions, memory management). Supports unmodified applications on top of guest OSes. Supports multiple different guest OSes with full isolation



Memory Management	
Segmentation	Cannot install fully-privileged segment descriptors and cannot overlap with the top end of the linear address space.
Paging	Guest OS has direct read access to hardware page tables, but updates are batched and validated by the hypervisor. A domain may be allocated discontiguous machine pages.
CPU	
Protection	Guest OS must run at a lower privilege level than Xen.
Exceptions	Guest OS must register a descriptor table for exception handlers with Xen. Aside from page faults, the handlers remain the same.
System Calls	Guest OS may install a 'fast' handler for system calls, allowing direct calls from an application into its guest OS and avoiding indirecting through Xen on every call.
Interrupts	Hardware interrupts are replaced with a lightweight event system.
Time	Each guest OS has a timer interface and is aware of both 'real' and 'virtual' time.
Device I/O	
Network, Disk, etc.	Virtual devices are elegant and simple to access. Data is transferred using asynchronous I/O rings. An event mechanism replaces hardware interrupts for notifications.

Table 1: The paravirtualized x86 interface.

- **I/O virtualization**: **Shared + Private pointers** between guest and Xen, Xen controls access to input devices, communicates to OSes through ring buffers. Requests have unique identifiers, and Xen does not guarantee a fixed order to process requests.
- **Time virtualization:** Realtime is how long Xen has been running (for networking); **virtual time** is how much time the guest OS has had to run; virtual time is used by guest OS to make its own scheduling decisions so it advances only when guest OS runs; guest OSes can set timers for real and virtual time. **Wall clock time** is the offset between Xen's "real time" and actual time.
- **Xen scheduling**: **Work conserving**; prefer recently executing OSes; prefer low latency switching.
- **CPU virtualization**: Guest OSes must be modified to run on a lower privilege level than Xen, all privileged commands must go through Xen.
- **Memory virtualization**: Xen allocates **physical memory** to guest OSes, ensures domains are partitioned and isolated. Guest OSes then create **page tables** to store virtual memory mappings and register them with Xen on creation. Guest OSes have read-only access to page tables, Xen must approve all updates.


Request Consumer
Private pointer in Xen
Request Producer
Shared pointer updated by guest OS
Response Producer
Shared pointer updated by Xen
Response Consumer
Private pointer in guest OS
▨ **Request queue** - Descriptors queued by the VM but not yet accepted by Xen
▨ **Outstanding descriptors** - Descriptor slots awaiting a response from Xen
▨ **Response queue** - Descriptors returned by Xen in response to serviced requests
☐ **Unused descriptors**

Andromeda:

Control Plane: **Cluster Management** - provisions networking, storage, compute on behalf of users; out-of-scope for Andromeda;
Fabric Management - handles aggregate configuration for cluster; delegates to VM and LB controllers actual instruction of what to configure; VM and and LB controllers talk to VM host switches to install rules;
Switch Layer - base on Open vSwitch receives programming from FM and applies OpenFlow to program datapath; Replicated controllers and OFEs guaranteed control plane reliability; No replication of switch layer (hosts deemed unreliable); Per customer queries in VMC with round robin processing;
Fail static - even after CP desceduled, it can function normally for a period of time; VM hosts sharded to be controlled by particular VM controller; leader and 2 followers; **Open Front Ends** - take information from VMC and translate it for VM host
Data Plane: main processor for packets;
Coprocessors are provisioned PER VM to handle some encryption, intrusion, analytics, etc.; **Fast paths** maintain cache of forwarding state and packet processing logic; On miss, packet sent to vswitchd which can update fast path state and re-inject packet
Routing: **Preprogrammed model** - consistent, predictable performance; doesn't scale; slow updates; **On demand** model - more scalable; high latency on initial packets; easily overwhelmed; **Gateway model** - consistent, predictable performance; control plane scales well (number of routes scale linearly); data plane doesn't (gateways can be overwhelmed by traffic); (**Andromeda**) **Hoverboard model** - combination of on-demand + gateway; data initially goes through gateway, popular routes have an on-demand route created (fast plane)
VM Migration: Hairpinning - packets sent to old destination until new destination is initialized; Buffers - store what packets must be sent and forward them to new location; Packets not received, TCP will try again
Flow Table: Lockup to map flow keys to flow indices; VM controller creates new flow table, swaps pointer so it looks at a different place; so we avoid conflicts in the flow table

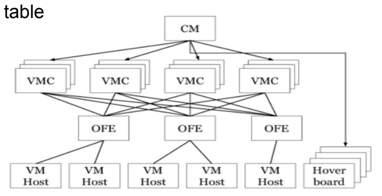


Figure 4: Control Plane Replication and Partitioning

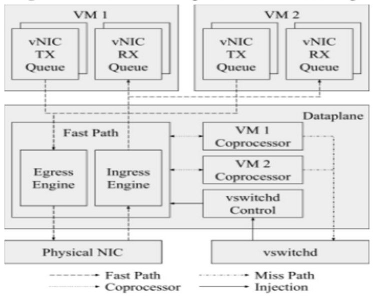


Figure 5: Host Dataplane Overview

Scheduling:
- **Time Sharing**: Time slicing, multiple tasks or users share the system resources, especially the CPU
- **Space Sharing**: Different tasks or users are allocated separate partitions or spaces of a resource
- **Long Term Schedulers**-starting new jobs on the computer. Often outside the OS;Non-preemptive;
- **Medium Term Schedulers**-Part of the OS. Decides if job should be swapped or suspended. Preemptive;
- **Short Term Scheduler**-AKA CPU Scheduler. Interleaves the CPU between the jobs that the other two have decided to run on the computer. Preemptive;State that needs to be saved in context switching: program counter;scheduling priority; all other registers;TLB/Page Table Information; I/O State information; Accounting Information;

- Iterative DNS Resolver: the client sends a DNS query to the resolver, and the resolver responds with a referral to another DNS server.
- Recursive DNS Resolver: DNS client relies on the resolver to handle the entire resolution process.